



SIGGRAPH 2022
VANCOUVER+ 8-11 AUG



Probe-based Lighting, Strand-based Hair System, and Physical Hair Shading in Unity's 'Enemies'

Francesco Cifariello Ciardi
Lasse Jon Fuglsang Pedersen
John Parsaie



Lighting and Hair in Unity's 'Enemies'

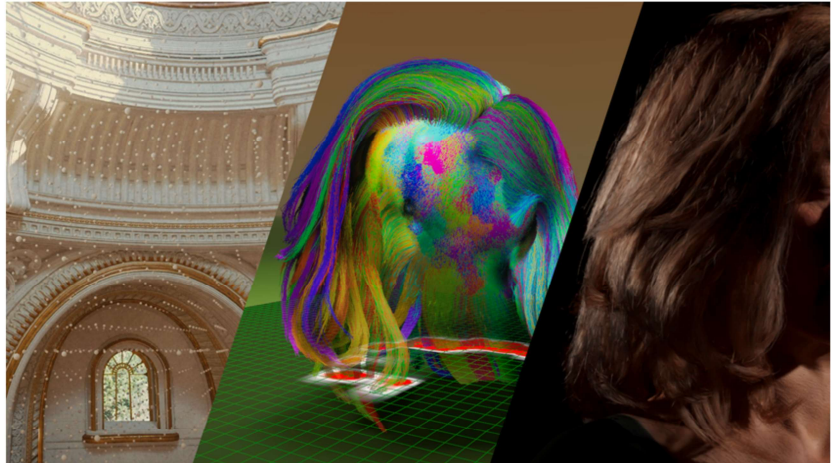
- Enemies is the latest real-time short film from Unity's internal Demo Team
 - Released earlier this year
- Large collaborative effort between teams
 - Production uses latest engine developments
 - Engine development guided by production
 - Improving the engine for our users





Lighting and Hair in Unity's 'Enemies'

- This talk will cover:
 - Probe-based lighting
 - Hair system/sim.
 - Hair shading





SIGGRAPH 2022
VANCOUVER+ 8-11 AUG

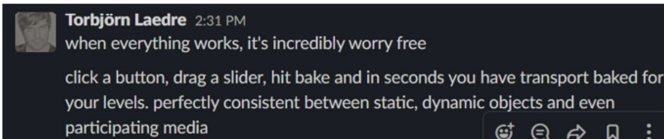


Probe-based Global Illumination

Francesco Cifariello Ciardi

Friends don't let 'Enemies' use Lightmaps

- Complex geometry w/ LODs
- Annoying authoring constraints
- Lack of unified lighting with character, the star of the film
- Worse directional quality
- ... We were developing a probe-based system so why not :D



Hi everyone I am Francesco and I am going to talk about precomputed indirect diffuse lighting in Enemies. Enemies doesn't use lightmaps. We all know why lightmaps might be problematic, complex geometry with many LODs are hard to deal with, they impose annoying authoring constraint, there is a lack of unified lighting with dynamic objects which is a problem given that the character here is the star of the film, and also we found we weren't getting the directional quality we wanted from Unity's implementation of directional light maps. Coincidentally, we were developing (and still are!) a replacement for the legacy per-object tetrahedralized probe system and therefore it felt like a good fit for the project.

Workflow ended up being much better as well... which is a great win!

The system doesn't do much especially new, but we stand tall on the shoulders of giants that preceded us. I am going to talk a bit about the choice that we made for this system before handing it over to Lasse and John to talk about Hair.



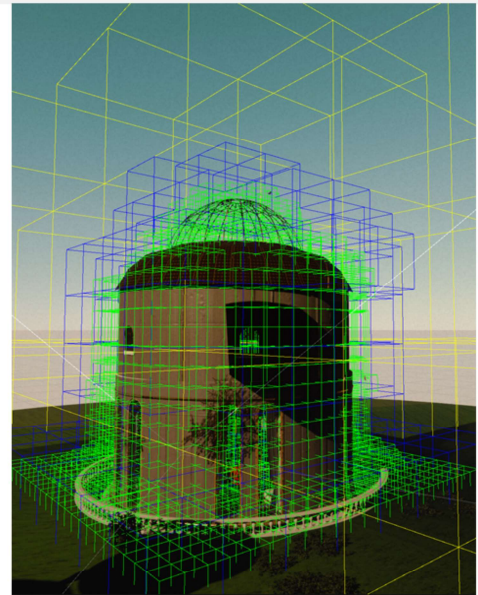






Adaptive Probe Volumes

- More density around geometry, coarse distribution farther away
- Artists can place volumes around the areas that will need placement of probes



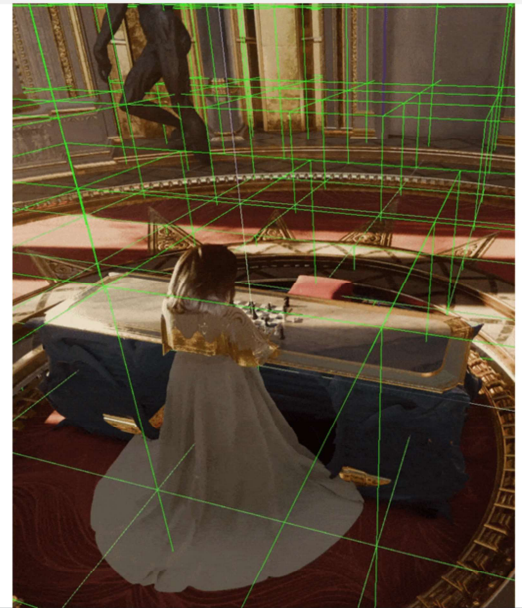
The system is adaptive in nature, we do an automatic placement step that adds probes near geometry with higher density, while doing a coarser distribution as we move farther away.

Artist will place probe volumes in the scene as markers of where the probes need to be placed, but it is important to notice the data is not owned by the volumes, it is instead slotted in a global structure.

CLICK

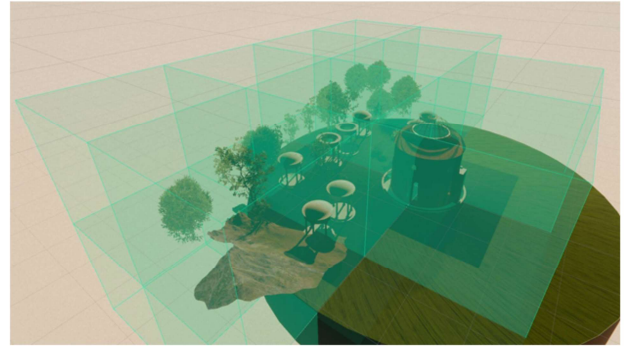
Adaptive Probe Volumes

- More density around geometry, coarse distribution farther away
- Artists can place volumes around the areas that will need placement of probes
- Parameters can be customized with local volumes



Volumes can also be placed to customize the probe density. For example a volume here is placed around the main character to have higher subdivision surrounding it as you can see by the red bricks.

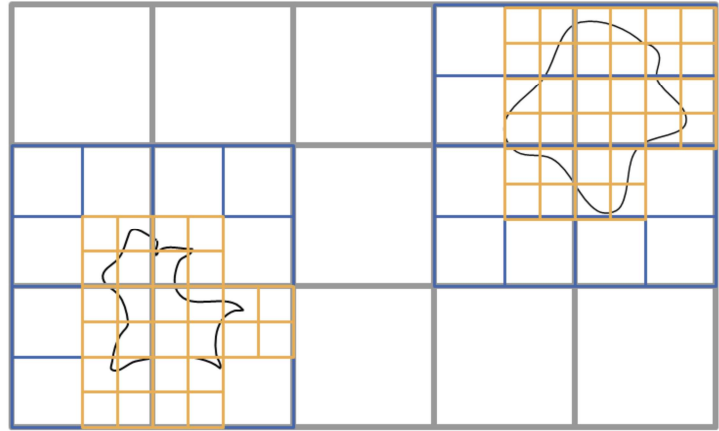
- We place probes in a minimum unit of 4x4x4 called brick
- Sets of bricks are grouped in “cell” units
 - Streamable and loadable unit
 - Streaming from disk to GPU passing through small CPU staging



For a bit of nomenclature, we place probes in units of 4x4x4 called bricks, corners between bricks of different subdivisions are shared so that for a given position we bake the data only once.

Bricks are grouped in what we call cells, those are none other than loadable and streamable units. I am not going to talk about streaming today, but the streaming happens from disk to GPU passing through a small CPU staging memory.

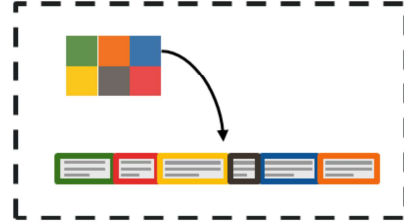
- Similar to [Garcia et al. 2020]
- We generate an SDF of the scene
- Guided by the SDF, we refine if we are close to geometry



**Note: In slides, the subdivisions divide by 2 only for illustration purposes, in the system is 3x.*

The actual placement of the bricks is very similar to what was presented for Need For Speed at SIGGRAPH 2020. We generate an SDF of the scene and [CLICK] guided by said SDF we are going to refine the subdivision if we detect we are close to surfaces.

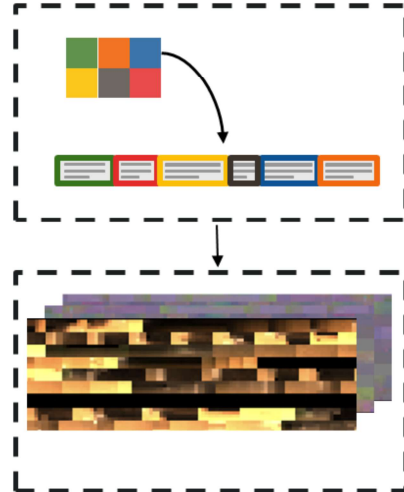
- World data structure has two parts
- An indirection buffer (World Pos to SH)
 - Global Cell indirection
 - Per-Cell Brick indirection



Now that we have our bricks, we need to sample the data somehow.

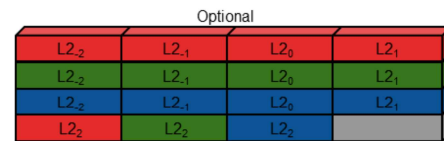
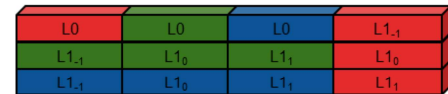
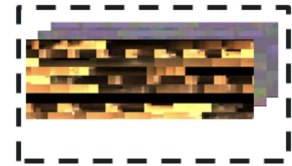
The data structure is made up of two parts, an indirection part that in itself has two building blocks: a global cell indirection structure and a per-cell brick indirection. [CLICK]

- World data structure has two parts
- An indirection buffer (World Pos to SH)
 - Global Cell indirection
 - Per-Cell Brick indirection
- Spherical Harmonics Pool



And we have a storage for Spherical Harmonics data that we call SH Pool.

- Fixed-Sized set of 3D textures
 - The depth is always 4 (the brick depth)
- Data for a single brick need to be contiguous, but no other spatial ordering constraints
- Spherical Harmonic data compressed as in [O'Donnell18]
 - L0: 16bit per channel (or BC6H)
 - L1 and L2: [0..1] values in function of L0 in 8bit per channel (or BC7)
 - Possible problems when filtering (see [Hobson19]), but worked fine for us



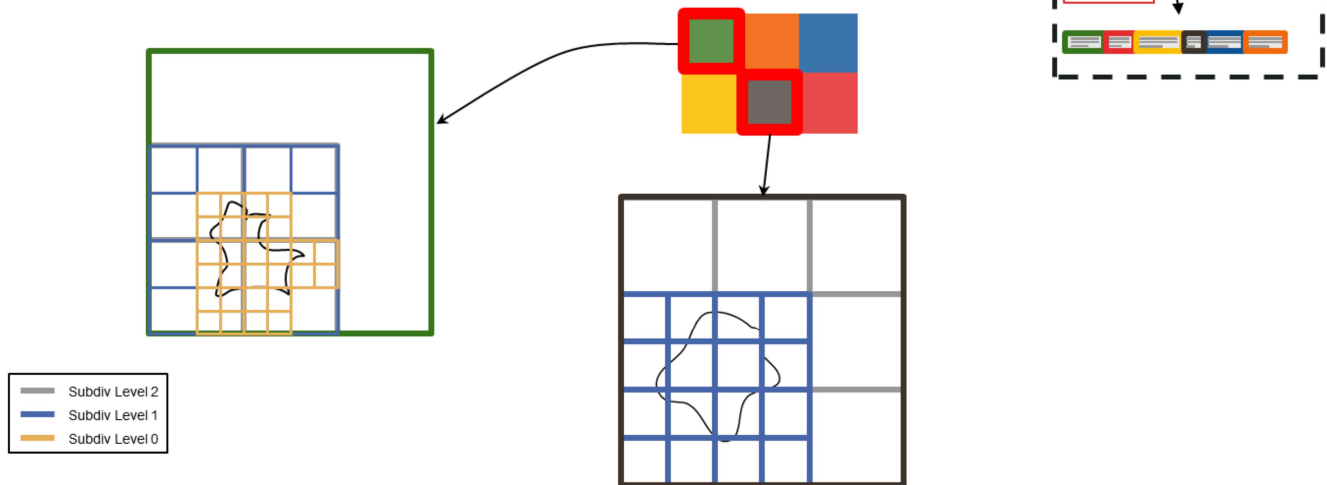
The SH texture pool is a set of fixed sized 3D textures where the only constraint is that a brick worth of data is contiguous (for filtering and indexing sake), but the rest of the data can be completely disjointed; there is no required knowledge of the placement location. The size of this texture is user-set according to memory budgets and it only determines how often we'll trigger a streaming event.

We store SH compressed as presented by O'Donnell in the Precomputed Global Illumination in Frostbite. L0 is stored with higher precision and L1/L2 are represented as numbers between 0 and 1 expressed in function of L0 and therefore that can be stored at lower precision. Note that L2 is optional.

This scheme was discussed as creating some problems by [Hobson] for god of war for filtering data. Those concerns are valid, and while it has been working fine for us we are keeping our eyes open nonetheless.

Data Structure - Indirection - Global Cell Indirection

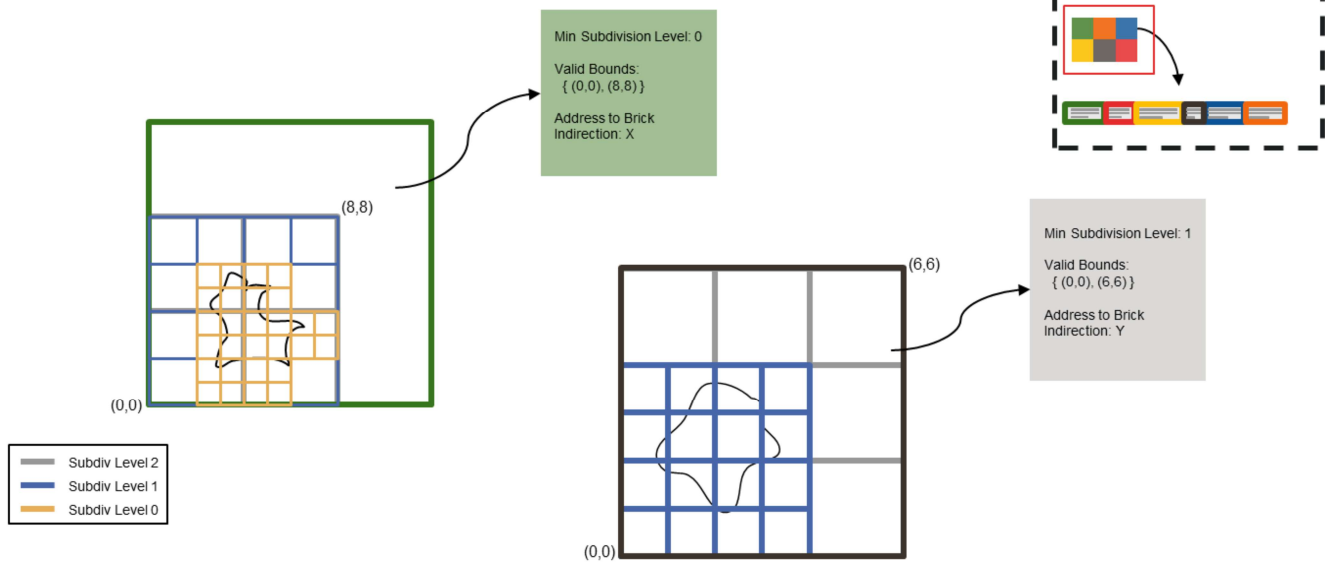
- Global data structure with an entry per cell (2 uints)



The first part of indirection is a data structure with an entry for each cell of the level or world. This data structure is global and always fully in memory. We don't expect many cells since those are generally fairly large, but even it were to be the case the data stored for each entry is small and heavily packed and therefore the impact in memory is manageable.

CLICK.

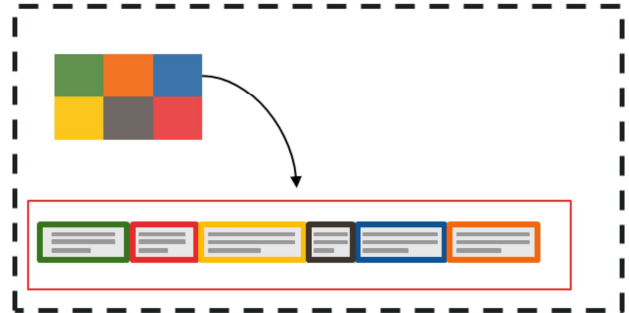
Data Structure - Indirection - Global Cell Indirection



Specifically what we store is:

- The highest density subdivision level
- The bounds of the area of the cell that actually contains bricks. And we represent these bounds in terms of minimum brick size
- And finally an address to the per-cell indirection buffer that will give us the actual mapping from world position to the SH data.

- Per Cell Indirection instead of global:
 - Can be shaped for the actual characteristics of the cells
 - Can be streamed together with the SH data

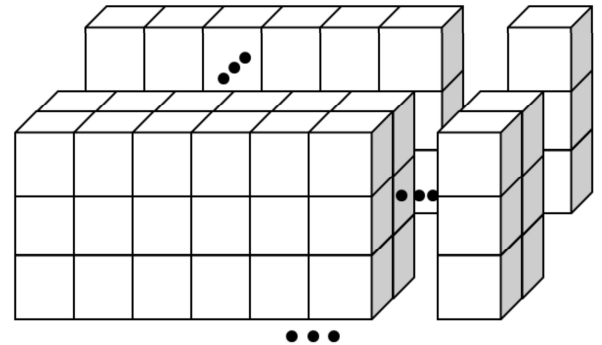
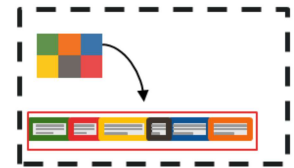


We keep this brick indirection per cell instead of having a global one for two main reasons:

- First, we can shape it for what is actually within the cell and we don't need to overallocate for the minimum common denominator. That means that if a cell contains only large bricks, the indirection structure will be small.
- More importantly, we can stream this part in and out without having to reupdate a monolithic structure every time; the sum of all these buffers might be a significant chunk of memory, so having the option to stream parts of it is really important.

- The brick indirection has one entry (1 uint) per brick in a cell assuming we might have a full cell at the highest density available

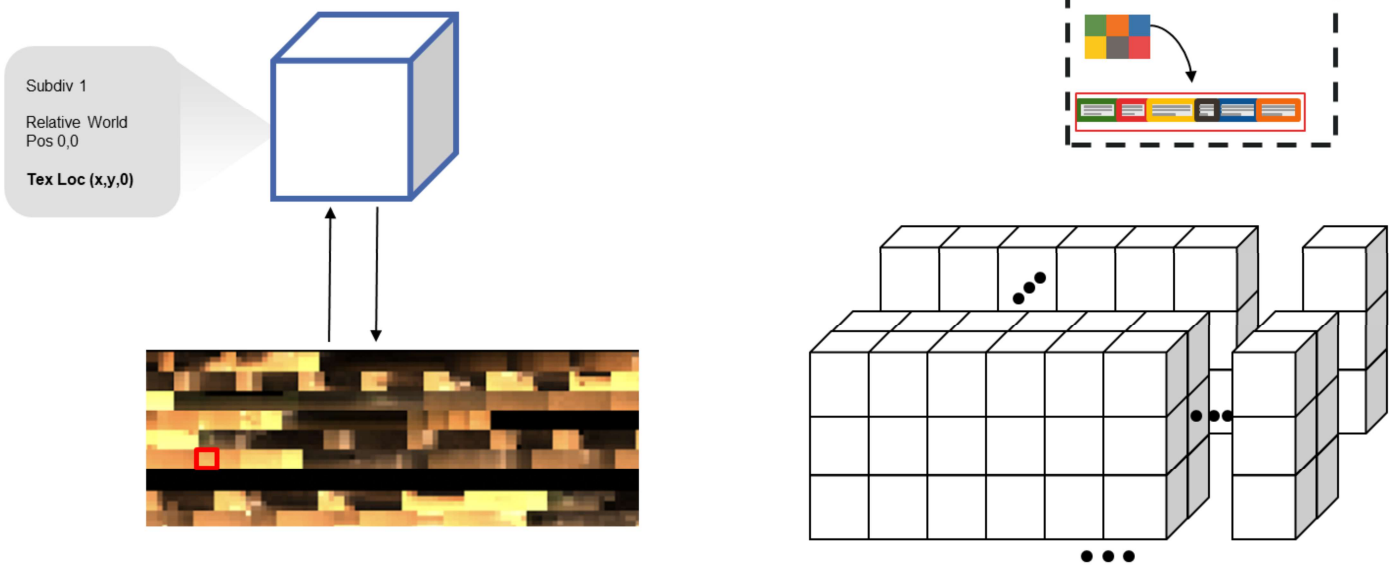
— E.g. Min brick size: 3m → Buffer will have 27^3 entries
Cell side size: 81m



This per-cell indirection buffer has an entry for each potential brick in the cell and it is built assuming we might have the area of the cell that is covered in bricks as if fully filled at the highest resolution available inside the cell. This will allow us to query the structure at the right resolution at any given point.

For example if we had a cell of 81 meter per dimension and a minimum brick size available in the cell of 3 meter, we will have 27 entries per dimension.

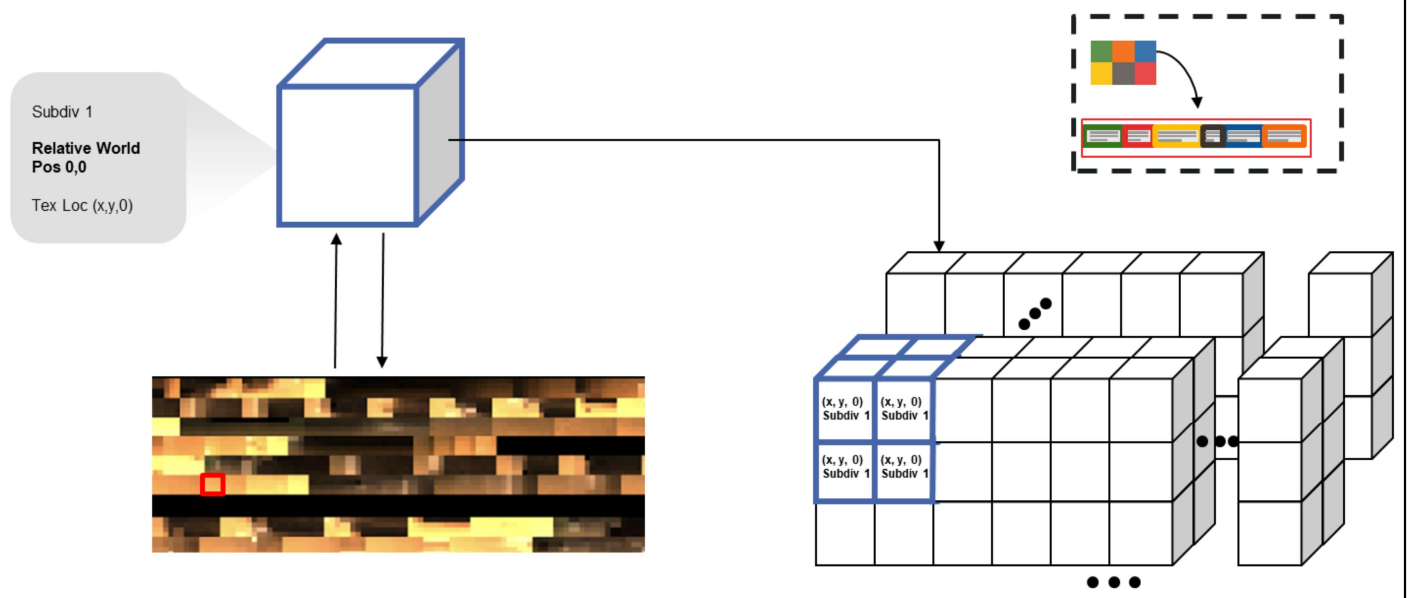
Data Structure - Indirection - Per Cell Brick Indirection



As we load a brick we place it in the SH pool. Upon placement we will note down the texture location where the placement has happened.

CLICK

Data Structure - Indirection - Per Cell Brick Indirection

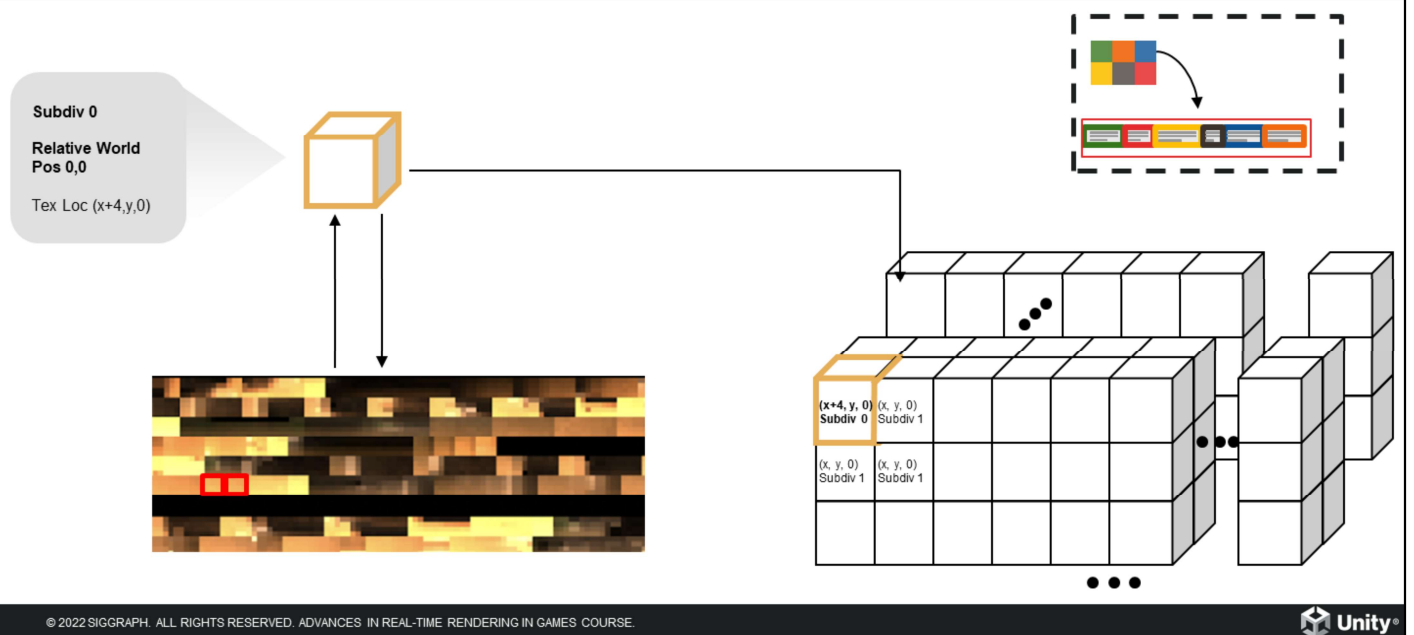


Using the world position of the brick relative to the cell origin, we can now place it in the indirection buffer.

For all the entries that overlap with the brick we insert the brick's texture location in the pool and its subdivision level.

Note how in this example the brick is of subdivision 1, and therefore NOT of the highest density we have available. For this reason, it span multiple entries that are minimum subdiv-sized and we will tag them all.

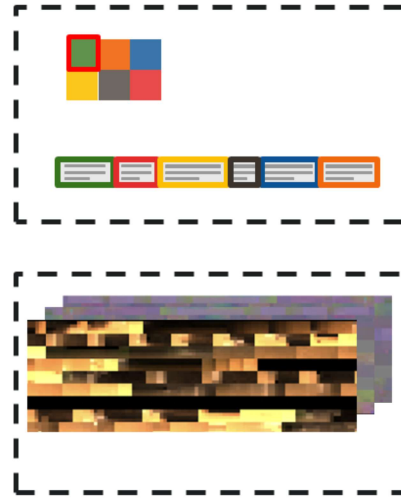
Data Structure - Indirection - Per Cell Brick Indirection



As a smaller bricks comes that is at the same world position, we replace the data previously stored from a brick at lower resolution, guaranteeing that we always store the coordinates to the highest quality data for any given entry.

```
int3 cellPos = floor(posWS / _CellInMeters);
float3 topLeftCellWS = cellPos * _CellInMeters;

cellPos -= _MinCellPosition;
// This samples the global cell indirection buffer
LoadCellIndirectionMetaData(cellPos, out int perCellBrickIndirectionAddress,
    out int minFoundBrickLevel, out AABB validArea);
// Find relative brick index
float3 relativePosWS = posWS - topLeftCellWS;
int3 localBrickIndex = floor(residualPosWS / (_MinBrickSize * pow(3, minFoundBrickLevel)));
localBrickIndex -= validArea.min;
// This loads the data from the Per-Cell Brick indirection structure
uint brickData = LoadBrickIndirectionData(perCellBrickIndirectionAddress, localBrickIndex);
UnpackBrickData(brickData, out int3 brickLoc, out float brickSizeAtLoc);
// Find the actual UVW
float3 brickUWV = (brickLoc + 0.5) * _PoolSizeInv;
float3 posRS = posWS.xyz / _MinBrickSize;
float3 offset = frac(posRS / brickSizeAtLoc); // [0;1] in brick space
offset *= 3.0 * _PoolSizeInv; // convert brick footprint to texels footprint in pool texel space
float3 uvw = brickUWV + offset;
...
APVSampleData shData = SampleAPV(brickUWV);
...
```



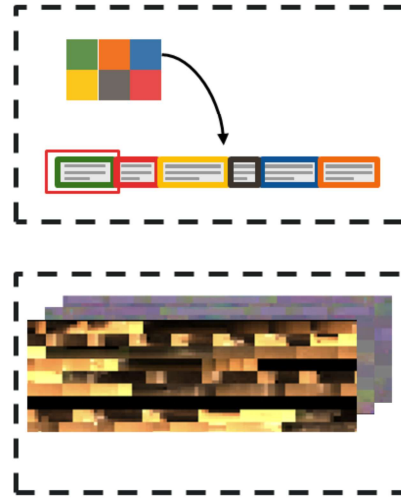
World Position → Cell Indirection → Per-Cell Brick Indirection → Brick UVW → Trilinear Sample SH Data

So to recap:

With a world position we identify the cell we are in and so the right entry in the cell indirection structure (it is just a matter of dividing by cell size and move relative to origin),
CLICK

```
int3 cellPos = floor(posWS / _CellInMeters);
float3 topLeftCellWS = cellPos * _CellInMeters;

cellPos -= _MinCellPosition;
// this samples the global cell indirection buffer
LoadCellIndirectionMetaData(cellPos, out int perCellBrickIndirectionAddress,
    out int minFoundBrickLevel, out AABB validArea);
// Find relative brick index
float3 relativePosWS = posWS - topLeftCellWS;
int3 localBrickIndex = floor(residualPosWS / (_MinBrickSize * pow(3, minFoundBrickLevel)));
localBrickIndex -= validArea.min;
// This loads the data from the Per-Cell Brick indirection structure
uint brickData = LoadBrickIndirectionData(perCellBrickIndirectionAddress, localBrickIndex);
UnpackBrickData(brickData, out int3 brickLoc, out float brickSizeAtLoc);
// Find the actual UVW
float3 brickUWV = (brickLoc + 0.5) * _PoolSizeInv;
float3 posRS = posWS.xyz / _MinBrickSize;
float3 offset = frac(posRS / brickSizeAtLoc); // [0;1] in brick space
offset *= 3.0 * _PoolSizeInv; // convert brick footprint to texels footprint in pool texel space
float3 uwv = brickUWV + offset;
...
APVSampleData shData = SampleAPV(brickUWV);
...
```



World Position → Cell Indirection → Per-Cell Brick Indirection → Brick UVW → Trilinear Sample SH Data

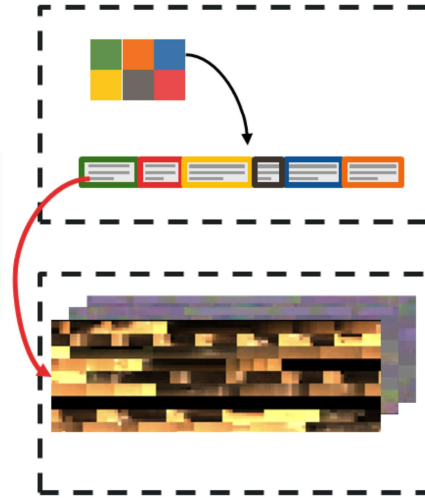
with the metadata in there we find the right per cell brick indirection for the cell we are sampling from CLICK

Runtime Sampling

```
int3 cellPos = floor(posWS / _CellInMeters);
float3 topLeftCellWS = cellPos * _CellInMeters;

cellPos -= _MinCellPosition;
// This samples the Global Cell Indirection Buffer
LoadCellIndirectionMetaData(cellPos, out int perCellBrickIndirectionAddress,
    out int minFoundBrickLevel, out AABB validArea);

// Find relative brick index
float3 relativePosWS = posWS - topLeftCellWS;
int3 localBrickIndex = floor(residualPosWS / (_MinBrickSize * pow(3, minFoundBrickLevel)));
localBrickIndex -= validArea.min;
// This loads the data from the Per-Cell Brick Indirection structure
uint brickData = LoadBrickIndirectionData(perCellBrickIndirectionAddress, localBrickIndex);
UnpackBrickData(brickData, out int3 brickLoc, out float brickSizeAtLoc);
// Find the actual UVW
float3 brickUWV = (brickLoc + 0.5) * _PoolSizeInv;
float3 posKS = posWS.xyz / _minBrickSize;
float3 offset = frac(posRS / brickSizeAtLoc); // [0;1] in brick space
offset *= 3.0 * _PoolSizeInv; // convert brick footprint to texels footprint in pool texel space
float3 uvw = brickUWV + offset;
...
APVSampleData shData = SampleAPV(brickUWV);
...
```

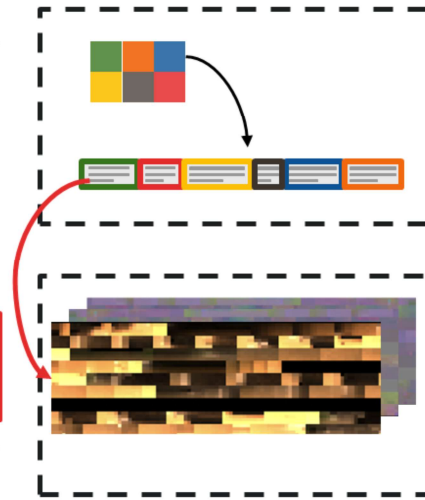


World Position → Cell Indirection → Per-Cell Brick Indirection → Brick UVW → Trilinear Sample SH Data

Now, we can move the world position to be relative to the cell and find the brick texture coordinates by sampling the per cell indirection buffer. CLICK

```
int3 cellPos = floor(posWS / _CellInMeters);
float3 topLeftCellWS = cellPos * _CellInMeters;

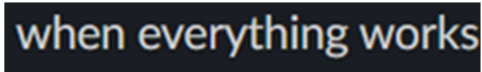
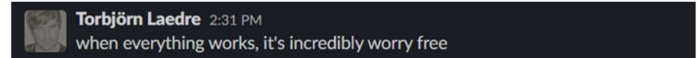
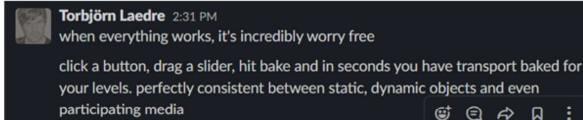
cellPos -= _MinCellPosition;
// This samples the Global Cell Indirection Buffer
LoadCellIndirectionMetaData(cellPos, out int perCellBrickIndirectionAddress,
    out int minFoundBrickLevel, out AABB validArea);
// Find relative brick index
float3 relativePosWS = posWS - topLeftCellWS;
int3 localBrickIndex = floor(residualPosWS / (_MinBrickSize * pow(3, minFoundBrickLevel)));
localBrickIndex -= validArea.min;
// This loads the data from the Per-Cell Brick indirection structure
uint brickData = LoadBrickIndirectionData(perCellBrickIndirectionAddress, localBrickIndex);
UnpackBrickData(brickData, out int3 brickLoc, out float brickSizeAtLoc);
// Find the actual UVW
float3 brickUWV = (brickLoc + 0.5) * PoolSizeInv;
float3 posRS = posWS.xyz / _MinBrickSize;
float3 offset = frac(posRS / brickSizeAtLoc); // [0;1] in brick space
offset *= 3.0 * _PoolSizeInv; // convert brick footprint to texels footprint in pool texel space
float3 uvw = brickUWV + offset;
...
APVSampleData shData = SampleAPV(brickUWV);
...
```



World Position → Cell Indirection → Per-Cell Brick Indirection → Brick UVW → Trilinear Sample SH Data

Finally, using the fractional world position within the brick we can craft the UVW used for our trilinear sampling.

THE PROBLEMS

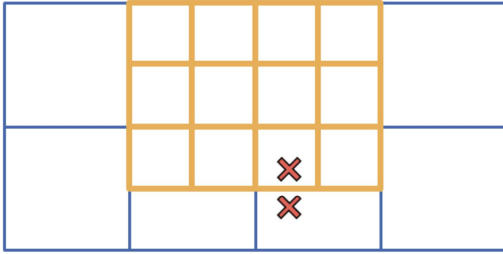


So this is the data structure, but of course we have problems!

Before I go on describing them, it is important to point out that we need to stay very lightweight as we need to support a large range of platforms. Moreover, since we are a generic engine we sadly cannot make any assumption on the content that the system is going to be used with.

Issues at Borders of Levels

- When sampling areas that jump from one level to another harsh seams occur

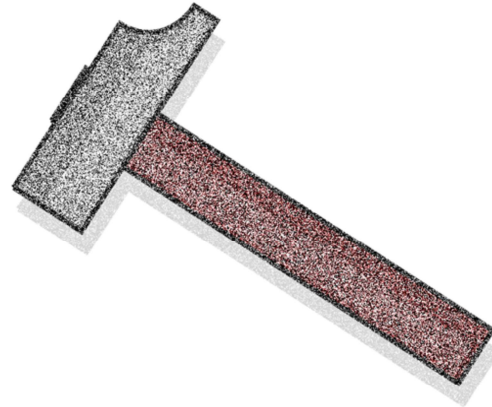


One of the issues that every multi resolution probe systems incurs into, is when neighbouring points in space sample from different resolutions. This leads to harsh strong seams as points that are close spatially will sample lighting data captured at different resolutions.

You can see the problem in this image highlighted in red

Well, CLICK .

- When sampling areas that jump from one level to another harsh seams occur
- Solve with an hammer: just add noise!
 - Animated Interleaved Gradient Noise applied in the local frame of sample position
 - Started as stopgap, it seems to work



We solve, or rather try to address, the issue with a delicacy of an hammer, a noisy hammer if you will!

CLICK

- When sampling areas that jump from one level to another harsh seams occur
- Solve with an hammer: just add noise!
 - Animated Interleaved Gradient Noise applied in the local frame of sample position
 - Started as stopgap, it seems to work

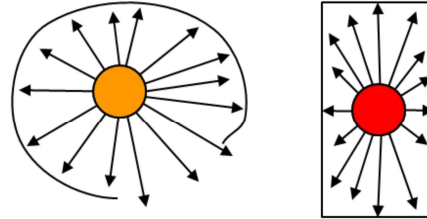


We apply animated Interleaved Gradient Noise in the tangent frame of the sample position, ending up breaking the straight seams into noise.

To be honest, this started as a crude stopgap solution, but it proved fairly effective and TAA smooths away most of the noise anyway.

Eventually, we ended up not observing the problem much especially on textured surfaces and given that this solution is very cheap, that's what we are going with at the moment, though I don't exclude that more effort will need to put in this.

- Probes can end up inside geometry
 - Placement is on a regular structure
 - Bricks are placed at fixed positions
- We assign probes a validity score based on what percentage of back-faces are seen during baking



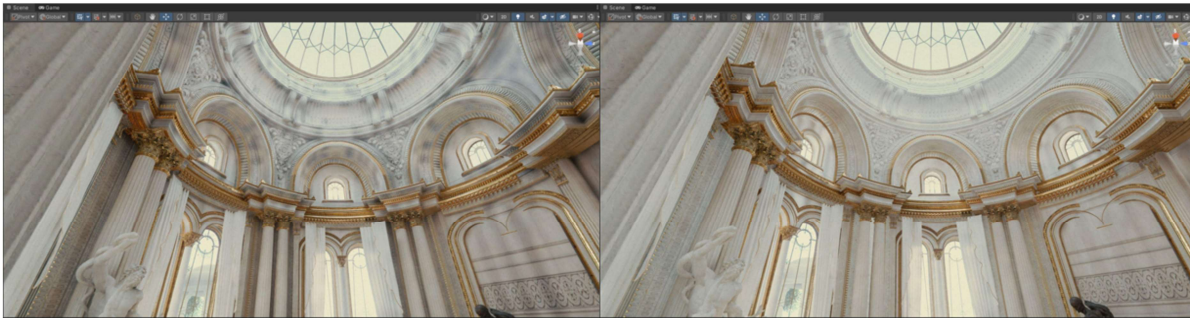
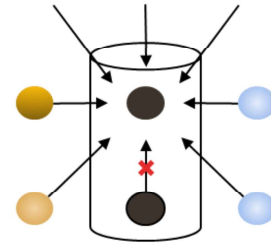
Another big set of issues come from probes that end up within geometry. This can happen fairly easily as bricks are a regular structure placed at locked positions in space.

CLICK We identify these problematic probes with a validity score based on how many backfaces are seen during the baking process.

If these invalid probes are left untreated they will leak black splotches throughout the scene, so we have to do something about it.

Dilation (Post-Bake)

- Dilate valid data into invalid probes
 - Weight based on inverse squared distance

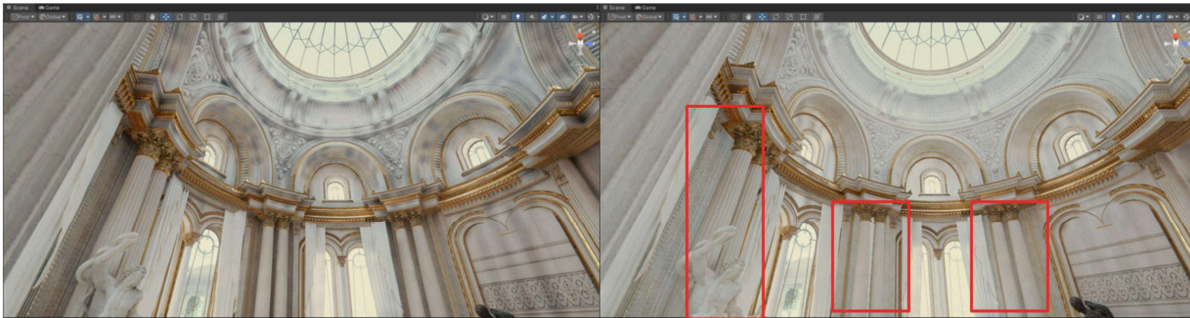
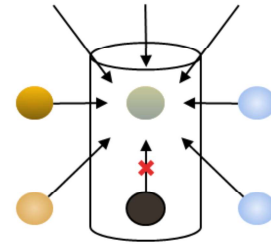


A first solution is dilation, this will happen immediately after we baked the data. Invalid probes gather data from valid neighbours (up to a distance set by user) and then will weigh those contributions based on the inverse squared distance from the probe we are dilating into.

CLICK

Dilation (Post-Bake)

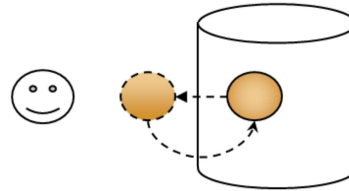
- Dilate valid data into invalid probes
 - Weight based on inverse squared distance
 - Can create leaking from other ends



As you can see by the images, with the raw output on left and post dilation on the right, this gets rid of a fair amount of black splotches. However, as you can see from the areas highlighted in red, it creates new leaks as for example data from outside gets diluted into probes sampled "inside". Here you can see that in the area behind the columns.

Again, sadly we can't make content specific assumption to guide this process better.

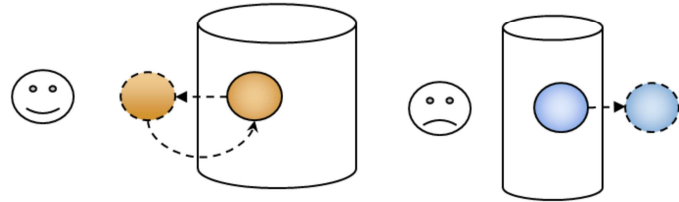
- Push invalid probes outside of geometry and bring them back in
 - Similar to [Caurant18][Garcia et al. 2020]



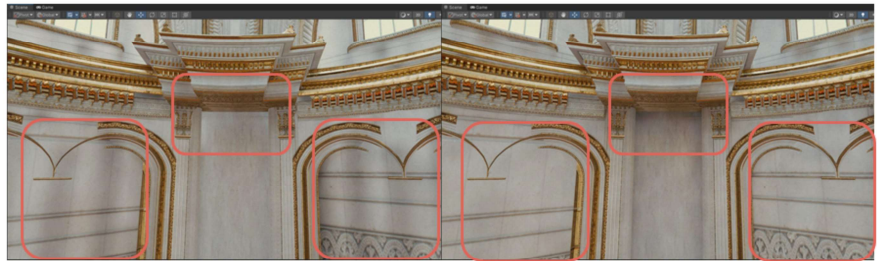
Another option is what we call Virtual Offset. This is similar to what was discussed by [Caurant for Detroit: Become Human and the Need for Speed presentation I mentioned before]. Essentially if a probe is inside geometry we try to find the closest way out, teleport the probe there for the sake of baking and then snap it back to the original grid position for the sake of sampling.

This works generally well, here is an example of when it does; you can see highlighted in red bright leaking on the top part of the column and the dark leaking from the wall are significantly reduced. However,

- Push invalid probes outside of geometry and bring them back in
 - Similar to [Caurant18][Garcia et al. 2020]
 - Can heavily skew capture point
 - Not always valid way out is found
 - This also can create other leaking



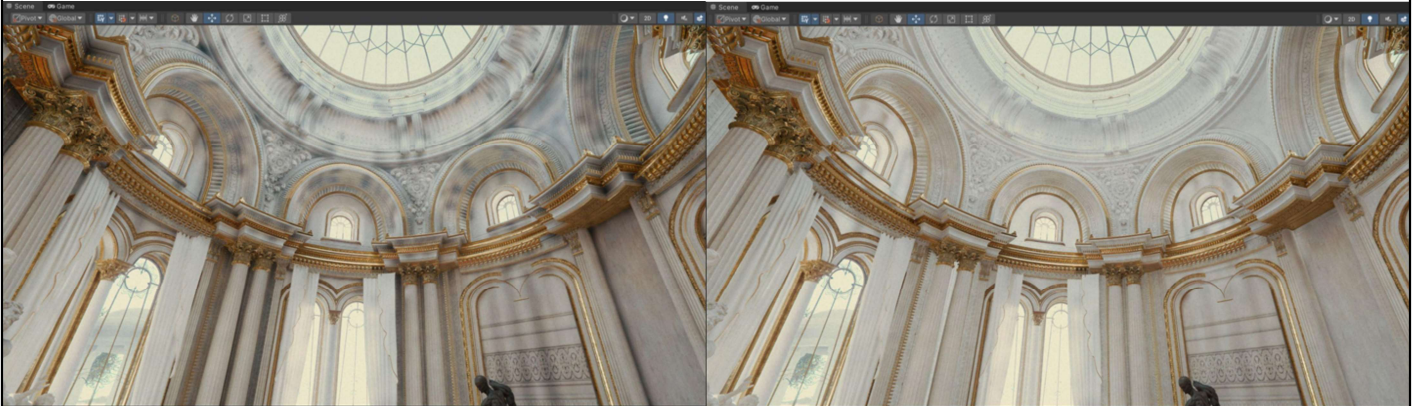
- Use both Dilation and Virtual Offset!



if used too aggressively, virtual offset it can skew the capture point too much and when we use it with judgement we have no guarantee that a way out is actually found. Also, it might create new leaking as the way out we find to bake might not be from the side we want or necessarily in a favourable spot.

In the end, both dilation and virtual offset work together. For the enemies Virtual Offset runs first and then for failing cases we run dilation.

Dilation only

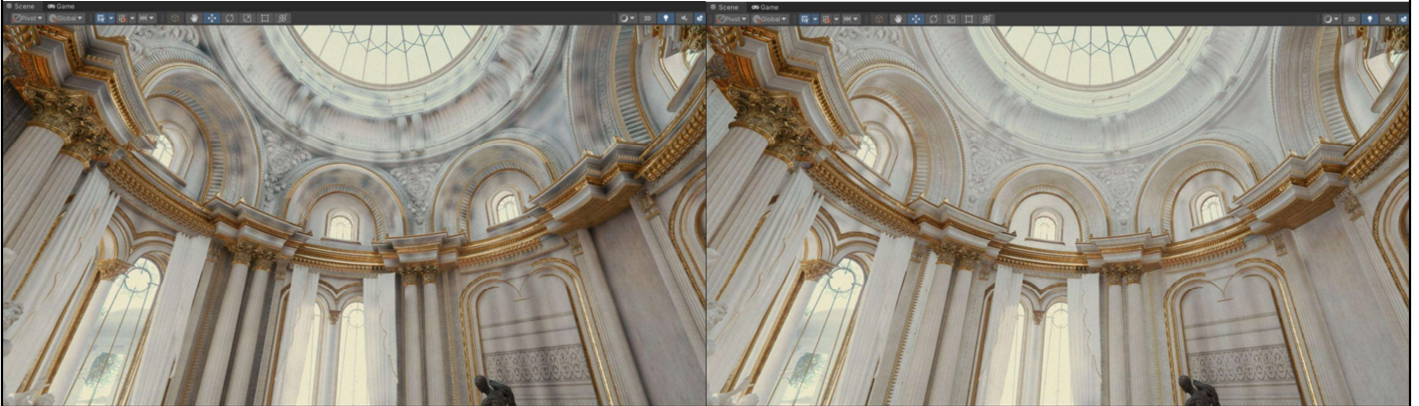


Raw Output

After Dilation

Let's see an image. With only dilation we can see how we get a significantly better result than the raw output, however as I mentioned before, some issues are still there and new leaks have been introduced.

Dilation + Virtual Offset



Raw Output

After Dilation and Virtual Offset

By running virtual offset first, some of the bright leaks are not there anymore, notice the area around the column [DO BACK AND FORTH]...
However a lot of problematic areas still are present, and some areas are arguably regressed.

So.. we need more!

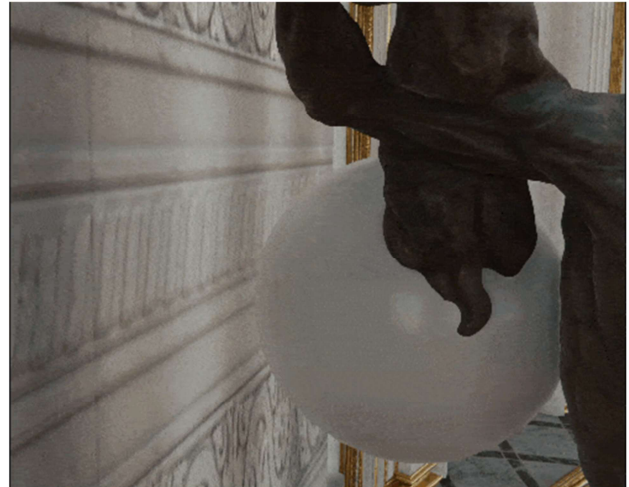
- At runtime we need to be lightweight
 - Cannot afford Octahedral Depth [McGuire19]
 - We accept we won't be perfect

We still have leaking from valid probes and the newly created leaks from trying to fix some black splotches. It is now time to do something at runtime.

But as I said before, we need to be lightweight, hence why we sadly cannot use incredible recent advances in leaking reduction such as the octahedral depth used in DDGI due to the additional runtime and memory cost.

We accept the results won't be perfect, but we will try our best nonetheless.

- At runtime we need to be lightweight
 - Cannot afford Octahedral Depth [McGuire19]
 - We accept we won't be perfect
- Biases
 - Normal Bias
 - Virtually inflates the sampling object
 - More discussions in [Hobson19]
 - Hard to tune, too much or too little can create problems



Firstly we start with some simple biases.

The first one is along the normal direction. This is very well discussed by Hobson, and I refer you to that excellent presentation, but essentially what it does is inflating the sampler to avoid self-sampling issues. This gets rid of several problems. However it is hard to tune. As you can see by the gif, pushing it too much shows weird data getting onto the sphere without any business to do so and too little will not be enough to account for our issues.

- At runtime we need to be lightweight.
 - Cannot afford Octahedral Depth [McGuire19]
 - We accept we won't be perfect
- Biases
 - Normal Bias
 - Virtually inflates the sampling object
 - More discussions in [Hobson19]
 - Hard to tune, too much or too little can create problems
 - View Bias
 - Favour data towards the camera
 - But view-dependent results
- Biases are powerful, but need to be used carefully and sparingly



The other bias we support is the View bias. This biases towards the camera, hopefully picking data that is more relevant to the viewer. Unfortunately as you can see by the gif, this produces a view dependent result which is sub-optimal.

Overall, both of course are just biases, and while they are powerful, they are annoying to tune and can lead to several problems. Generally we find that is very important to have them, but it is equally important to keep them at the minimum value acceptable to avoid creating issues.

So... yet again... we need to do more!

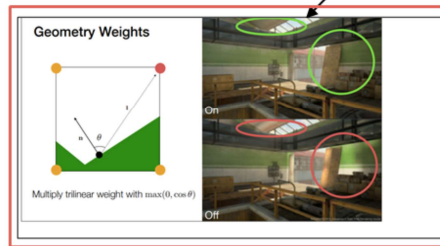
Runtime - Warping of Trilinear Sample Location

- Need to keep 1 single trilinear sample, so we warp the UV with some weight
 - Geometry weighting [Silvennoinen and Timonen 2015]
 - Validity based

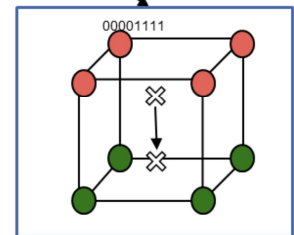
```
for (int i = 0; i < 8; ++i)
{
    float trilinearW = GetTrilinearWeight(...);
    float normalW = GetNormalWeight(...);
    float validityW = GetValidityWeight(...);

    weights[i] = saturate(trilinearW * normalW * validityW);
    totalWeight += weights[i];
}

for (i = 0; i < 8; ++i)
{
    newFractionalCoord += GetSampleOffset(i) * weights[i] * rcp(totalWeight);
}
```



[Silvennoinen and Timonen 2015]



While it is true that we need to keep only a single hardware trilinear sample per texture for performance reasons, we can warp the trilinear sample location by modifying the trilinear weights with some additional weighting. Specifically we use two schemes,

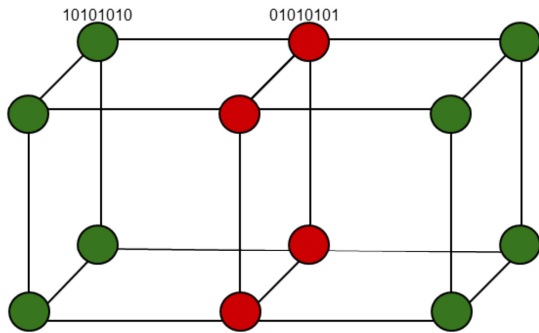
- A Geometry weight inspired by Remedy's/Silvennoinen
- And a validity based method I am going to discuss next.

- Dilation and Virtual Offset get rid of black splotches, but can introduce extra leaking
- We want to avoid sampling invalid probes altogether

We treated our invalid probes with dilation and virtual offset, and while those processes do an overall good job, as we have seen they can also introduce new issues.

More generally, we just want to avoid sampling data from probes that are inside geometry. Either they represent an occluding mesh or they might contain newly leaky data.

- Each probe stores an 8bit mask, each bit tells if a probe in 2x2x2 neighbourhood is invalid (i.e. validity score below a certain threshold)
- If a neighbour is invalid, push sample position away from it

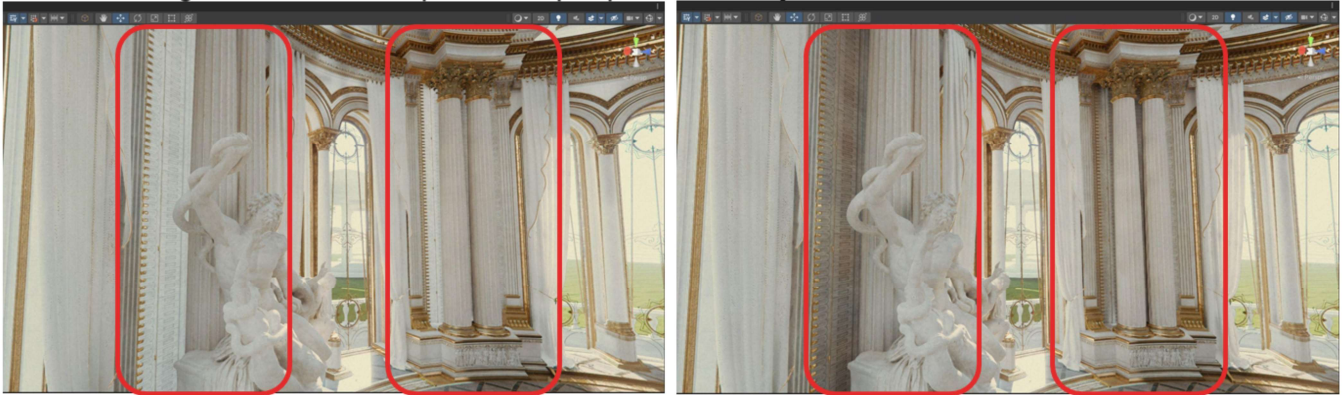


```
uint neighbourhoodMask = Sample(MaskTexture, UVW);
for (int i = 0; i < 8; ++i)
{
    int neighbourBit = 1 << i;
    int isNeighbourValid = (neighbourhoodMask & neighbourBit);
    // This is binary in our case
    float validityWeight = GetWeight(i, isNeighbourValid);
    ...
}
```

To avoid sampling them, at bake time each probe creates an 8bit mask where each bit represent a probe in the 2x2x2 neighbourhood used for sampling. A bit in this mask is set to 1 if the corresponding neighbour is valid and is set to 0 otherwise. Using this info, we can compute a weight that we can use to push the sample location away from the invalid probes. In our case it is just a binary one.

Validity Based Weighting

- Each probe stores an 8bit mask, each bit tells if a probe in 2x2x2 neighbourhood is invalid (i.e. validity score below a certain threshold)
- If a neighbour is invalid, push sample position away from it



© 2022 SIGGRAPH. ALL RIGHTS RESERVED. ADVANCES IN REAL-TIME RENDERING IN GAMES COURSE.

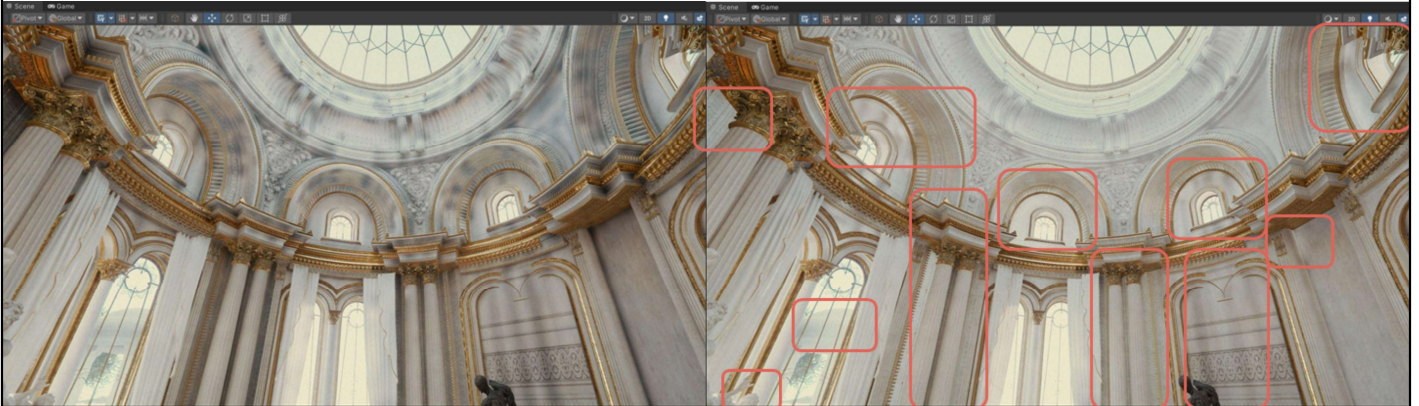


This of course adds a single extra 8bit texture load, but the visual gains are significant enough.

As you can see from the image and specifically the parts highlighted in red, we manage to avoid some of the leaking dilation introduced.

Note that for this image the geometry weighting has been turned off to show the impact of the validity weighting alone.

Dilation + Virtual Offset

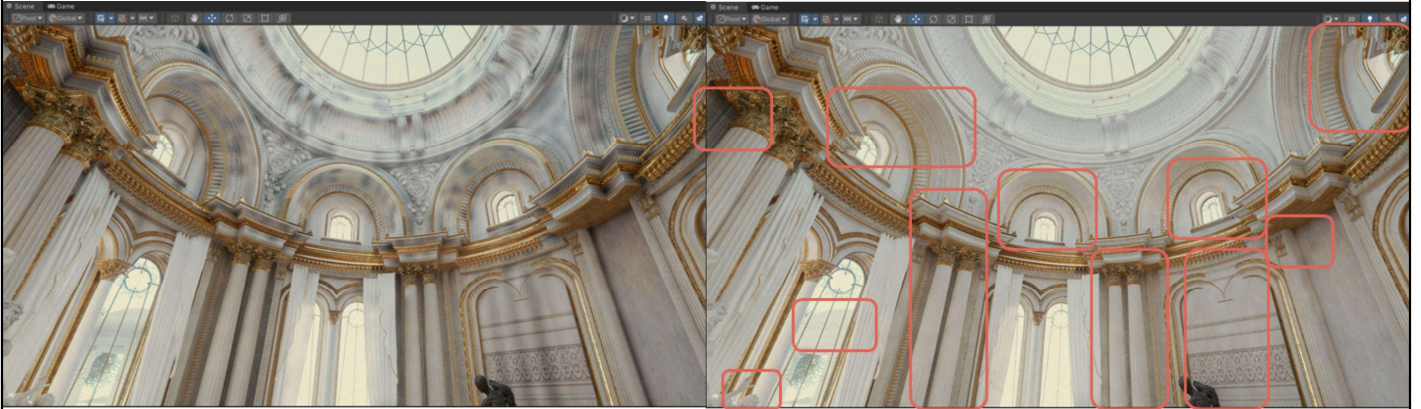


Raw Output

After Dilation and Virtual Offset

So let's go back to the image I showed you before.
Dilation and virtual offset give us a decent start, but lots of problematic areas are still there, for example the ones I highlighted in red. [CLICK](#)

Dilation + Virtual Offset + Weights

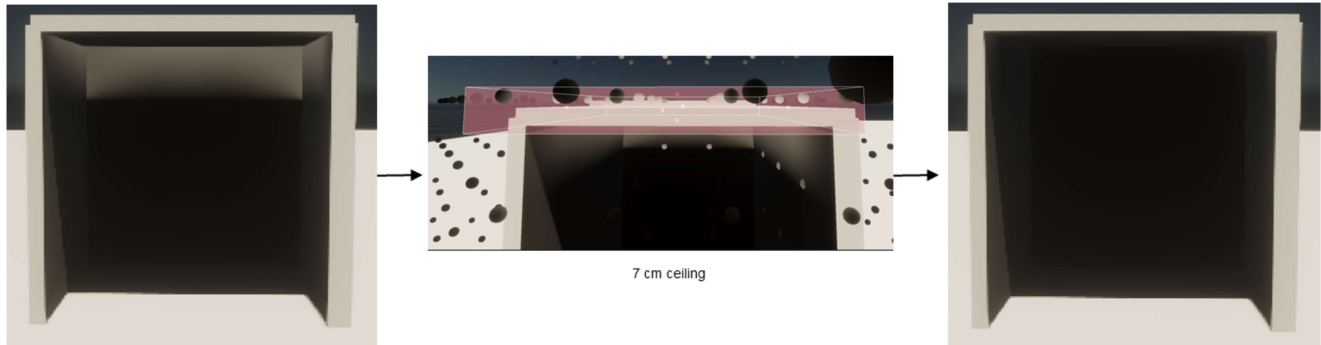


Raw Output

After Dilation, Virtual Offset + Sampling Weights

Using our weighting we can see how many of those problematic areas are now fixed. [DO BACK AND FORTH]
Note that in these images biases are set to an extremely small epsilon and yet we still get a fairly clean image.

- Probe sampling rate for validity is not always enough
 - No guarantee probes fall within geometry at interface of possible leaks
- Allow to manually set probes falling within a volume to invalid
 - Can create virtually thicker walls for the sake of the leaking prevention



Now, of course this validity weighting relies on the fact that our probes will sample the scene accurately, more specifically that we will always have probes falling within the geometry at the interface of a possible leak.

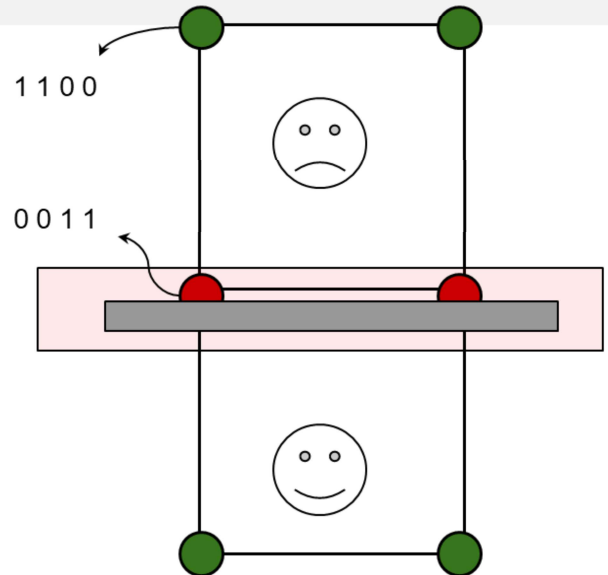
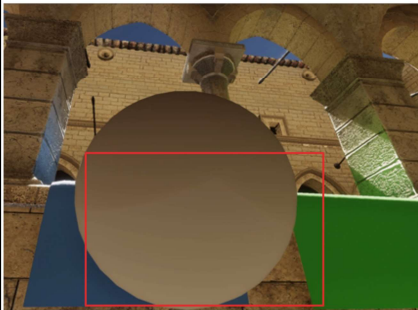
This of course cannot always be the case. So, we offer a primitive volume that artist can place and everything falling within it will be marked as invalid for the sake of this weighting scheme.

This essentially is the same of virtually increasing the size of specific objects like the ceiling of this box that had walls of only 7cm thickness. As you can see by invalidating the probes as if they were inside the ceiling, the leaking is heavily reduced.

I was very worried that this manual intervention would put off the artists. However, it turned out that this placement is not needed often, in fact for Enemies the feature is not used at all. Given that this is not a mandatory step, our artists actually quite welcomed this new degree of control it give them.

Touch-up Volumes for Invalidation

- This can lead to pushing away when not needed



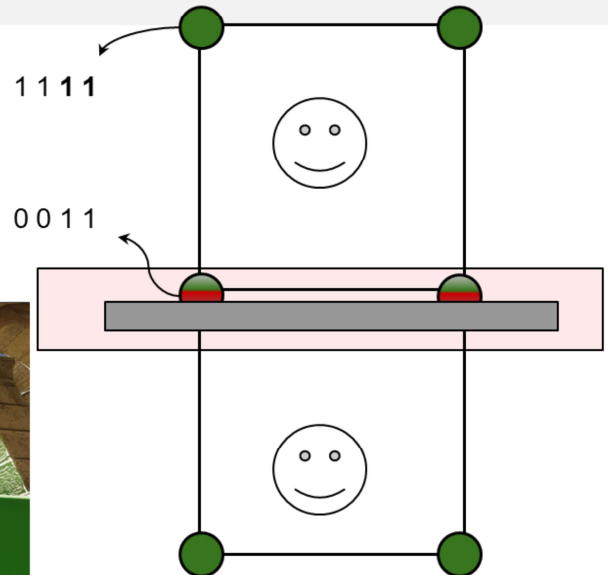
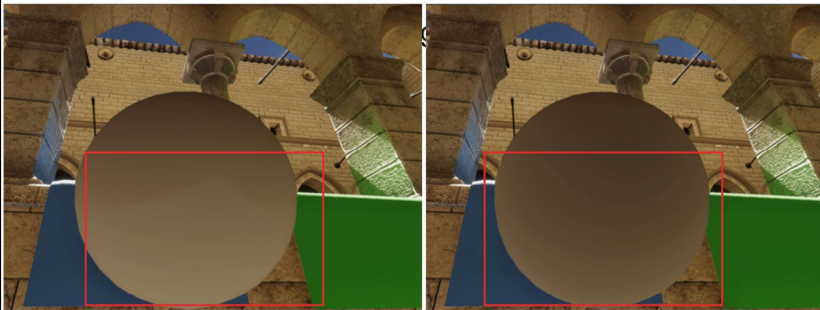
But we must not forget that a sampler probe trash is another sampler probe treasure!

Consider the example, a touch up volume invalidated a couple of probes. Most points sampling the bottom 2x2 blocks of probes is going to be happy with the invalidation as an occlusion is actually there made by the grey box; however the top neighbourhood has no need to push the samples away from those probes as that neighbourhood is free of occlusion and those probes have been invalidated only because of the touchup volume that is placed for the grey box.

See for example the sphere image here, an artifact is visible on the bottom of the sphere due to this problem.

Touch-up Volumes for Invalidation

- This can lead to pushing away when not needed
- We check if the 2x2x2 neighbourhood is empty, if so, it means no need to invalidate
 - Restore the bits as valid



To partially account for this, if we detect that there is no occluding objects in the 2x2x2 neighborhood, we restore the validity in the mask used for computing the weighting.

Now the top neighbourhood in the example can have all the bits set to valid and freely gather data from all probes, while the bottom one can keep them invalid and push away from those.

Of course this is not perfect, but it is better than not doing it.

- Touch-up volumes evolved to influence much more:
 - Modify the Dilation process (e.g. validity threshold in area)
 - Modify the Virtual Offset process (e.g. change the push out distance)
 - Restore probe validity
 - Intensity scale
 - I swear I was forced.

I have run out of time, so I cannot discuss here how the touchup volumes evolved into becoming much more than this. We started experimenting with using them as also a mean to guide a bit the dilation and virtual offset process with localized content specific adjustments for example, a way to restore probe validity in the rare cases in which we might want to ... and we also added a last resort intensity scale that I was forced at gun point to add... but it is very well hidden I swear!

Also I haven't discussed many other features of the system, such as reflection probe normalization and sampling from volumetric fog.

It is now time to pass on to Lasse to talk about a more hairy subject

- Touch-up volumes for Dilation:
 - Change the validity threshold after which a probe requires dilation
 - Change how far the probes are looking for neighbours to grab valid data from
 - Bias the direction of dilation (i.e. grab data more on one side vs equally from all directions)
- Touch-up volumes for Virtual Offset:
 - Change how much an invalid probe is pushed outside of the geometry
 - Force an hard-coded virtual offset direction.

Reflection Probe Normalization

- Very similar to [Lazarov13]
 - Add artist-facing control to shape the normalization factor
 - Artists required control to have normalization only to be darkening rather than brightening the reflections
- Very valuable in reducing the reflection probe density requirements.



- At bake time we can assign a label to baking results
- As long the probe placement is compatible we allow to blend between scenarios
 - Can be used to fake time of day between keyframes



- Only SH data is stored per scenario, the auxiliary data is shared
- At runtime we blend a user-set amount of cells as requested by user input using a small temporary SH pool
 - Cells chosen with same criteria as streaming
 - Temporary pool then copied into the relevant areas of the main SH pool
- If the feature is used we don't use block compression on the SH pool used at runtime
 - Considering on the fly GPU compression

- Memory cost (can be reduced with more careful placement):
 - Global Cell Indirection: 80 bytes
 - Sum of all Per-Cell Brick Indirections: 300kb
 - SH data costing roughly 3MB (L2) without texture compression.
- Runtime cost
 - At the time of writing the slides Enemies did not run on consoles yet, so data taken from a test scene of similar characteristics
 - On PS4 Base (delta vs no probes or any other GI):
 - L1: 0.3ms
 - L2: 0.5ms
 - Performance numbers vary depending on how many cells are accessed due to cache behaviour.
 - We still have room for optimizing further



SIGGRAPH 2022
VANCOUVER+ 8-11 AUG



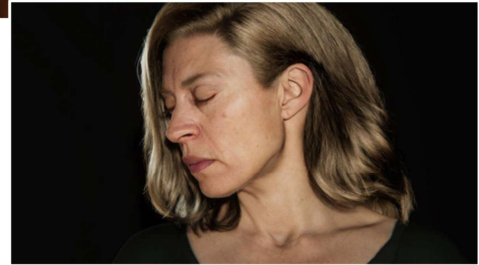
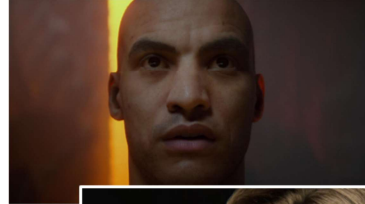
Strand-based Hair System

Lasse Jon Fuglsang Pedersen

@codeverses

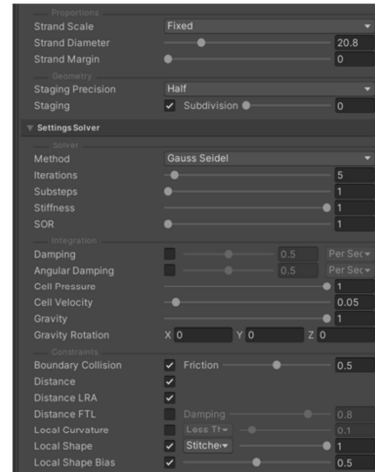
Strand-based Hair System

- A work in progress since 'The Heretic'
 - We used just stubble back then
- We needed a solution for 'Enemies'
 - Actress had long flowing hair
 - Wanted workflow for rapid iteration
 - No baking
 - Real-time simulation
- Encouraged by other teams' results
 - Frostbite video



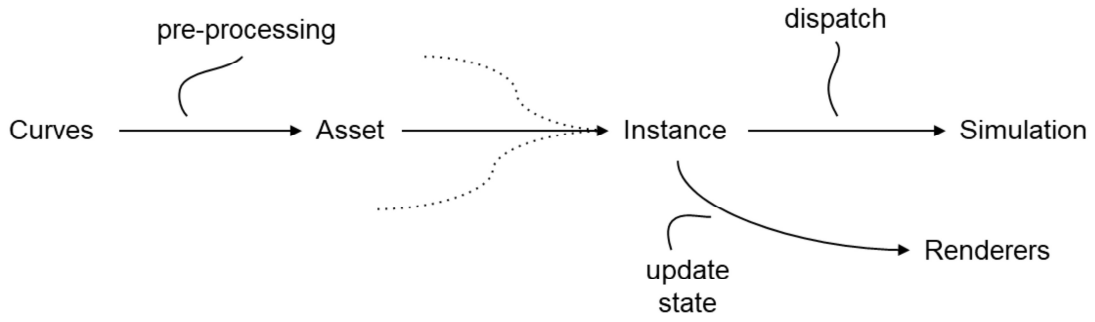
Strand-based Hair System

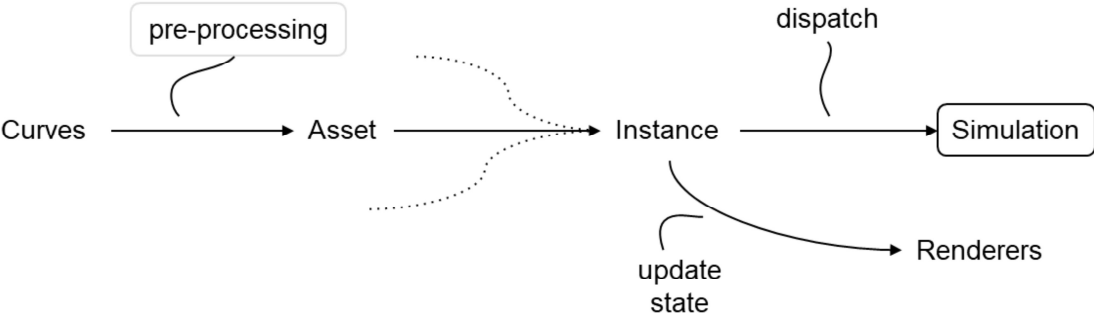
- No prior solution offered by Unity
 - Let's build something nice
 - Something complete and user-facing
- Requirements grew with work on the demo
 - "Long hair to match the actress"
 - "Adding some curls sure would be nice"
 - "We're going to need to preserve clumps as well"
- Led to configurable system
 - Toggle features based on application needs



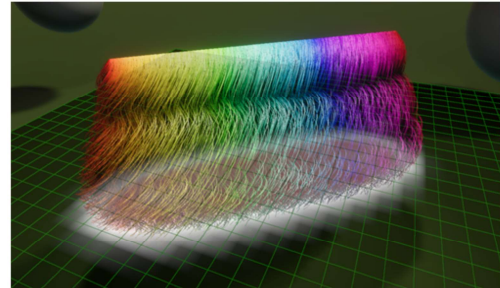
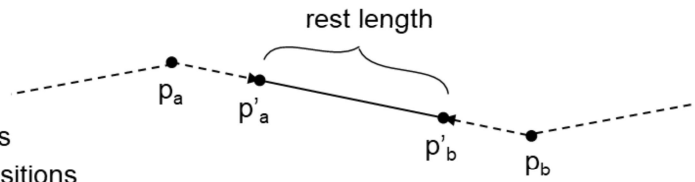
<https://github.com/Unity-Technologies/com.unity.demoteam.hair>

Overview

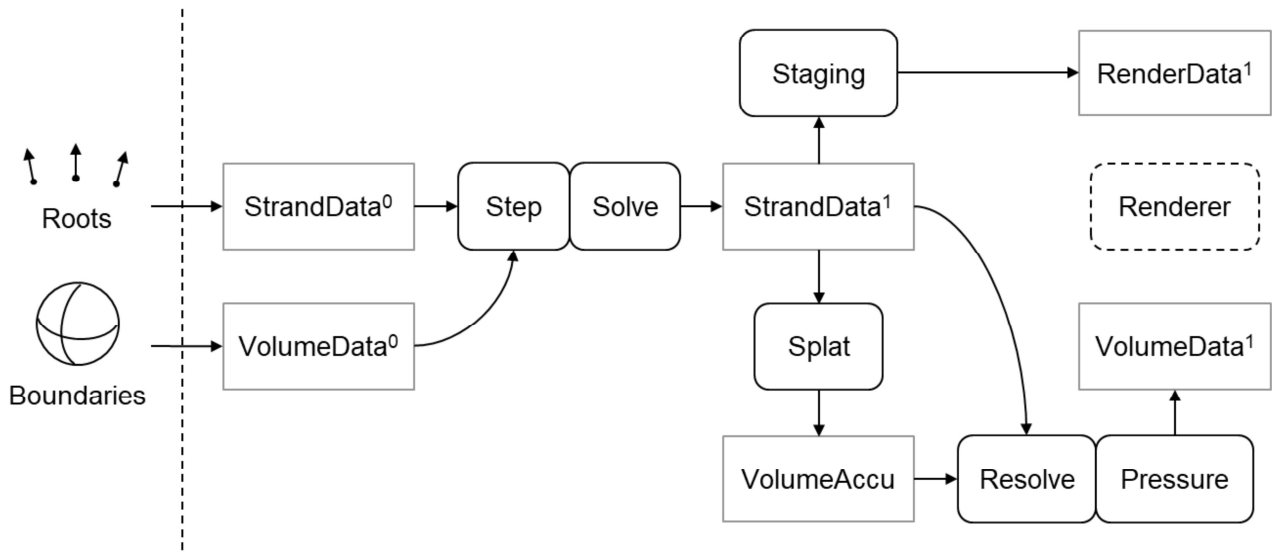




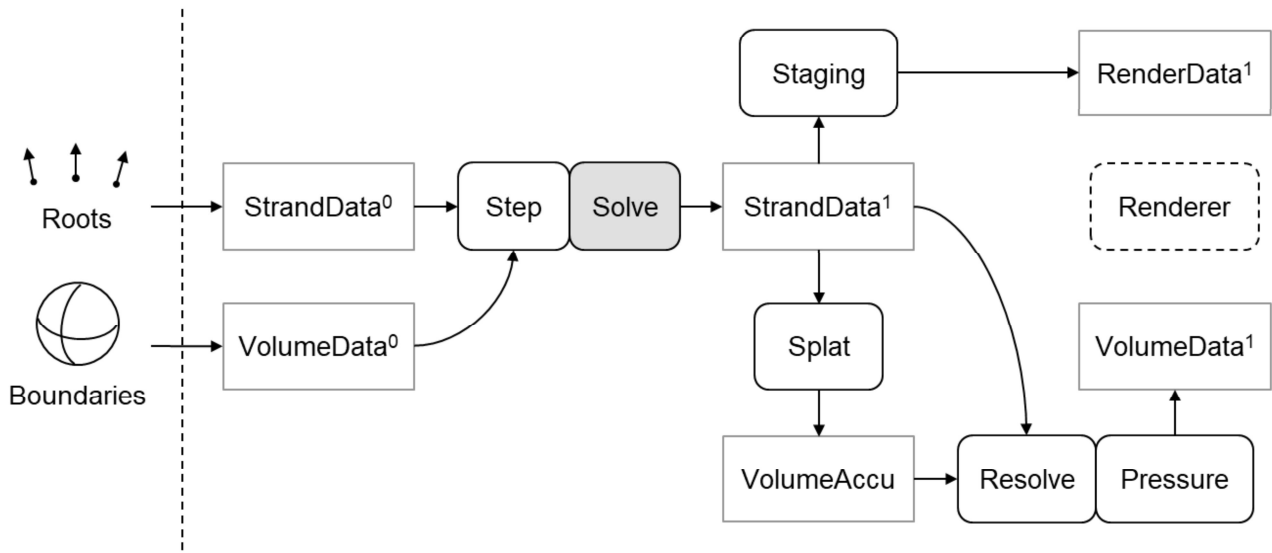
- Iterative strand solver
 - Regular PBD [Müller et al. 2006]
 - Update positions to satisfy constraints
 - Update velocities from changes in positions
- Volume data
 - Splat particles to grid [Petrovic et al. 2005]
 - Iterative pressure solve [Harris 2004]
 - Poisson equation with divergence on rhs.
 - Modify divergence to include source term
 - Gives us density control and soft hair-hair interaction similar to [McAdams et al. 2009]



Simulation Frame

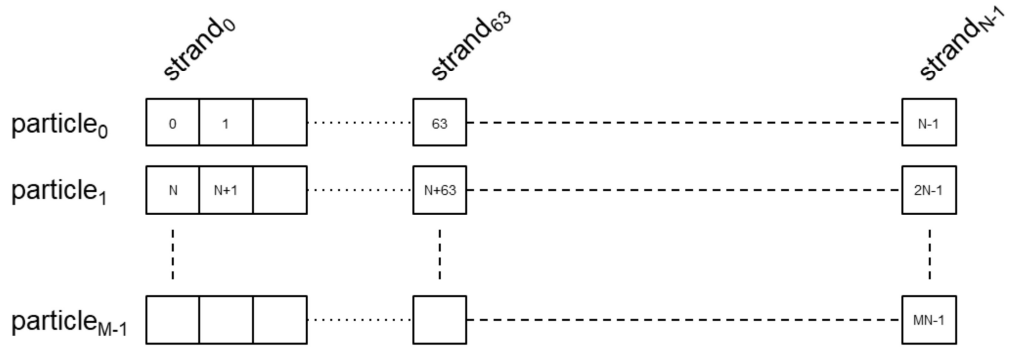


Simulation Frame



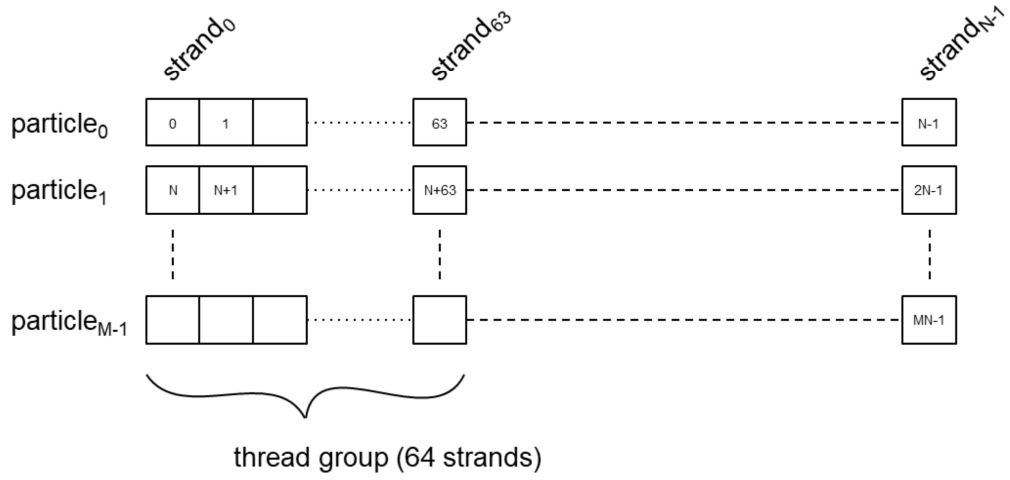
- Strands are pre-processed for the solver to assume certain things
 - Ordered by earliest LOD (that they are primary strands in)
 - Uniform number of particles per strand within group
 - Uniform particle spacing within strand
 - Particle data interleaved
- Particle buffers
 - Position, updated per iteration
 - Velocity, updated per timestep
 - Rest pose root offset, written on init
 - Rest pose frame delta, written on init

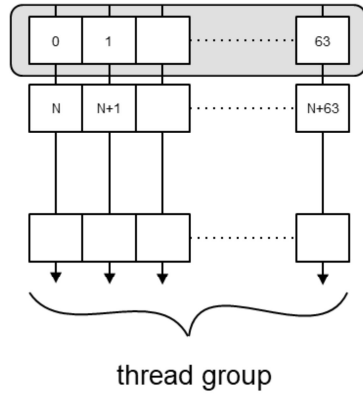
Strand Data



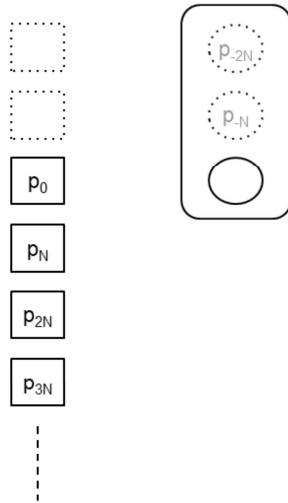
M particles per strand, N strands

Strand Data



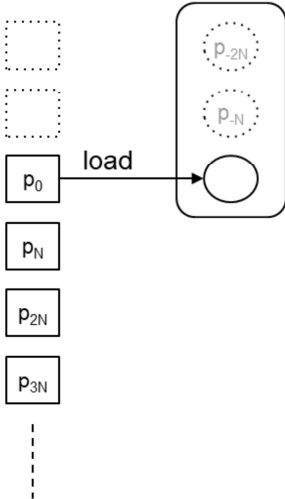


there are only few divergent paths – decent chance they'll stick together

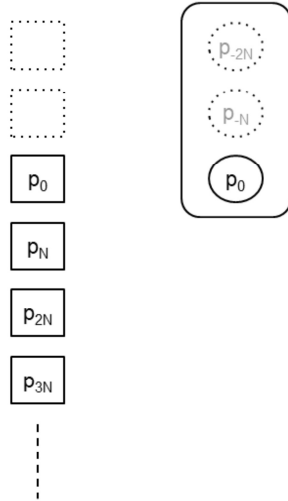


- Constraints are solved within a window
 - System only dealing with strands
 - Relationships are implicit
- Solver window
 - Initialized below the root of the strand
 - Moves down strand

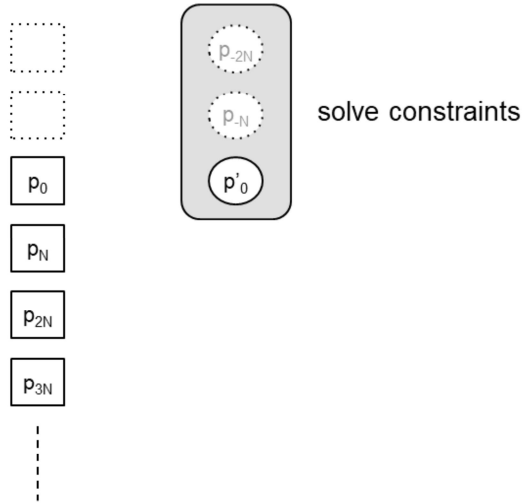
Constraint Solve



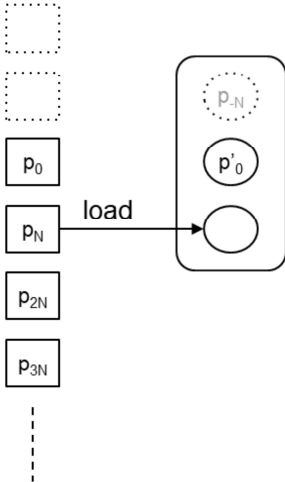
Constraint Solve



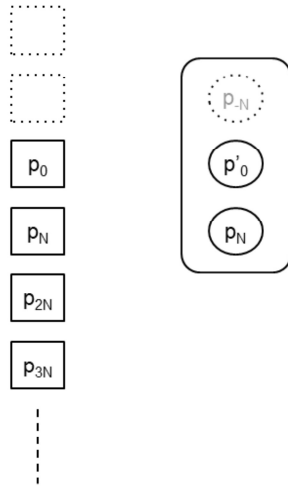
Constraint Solve



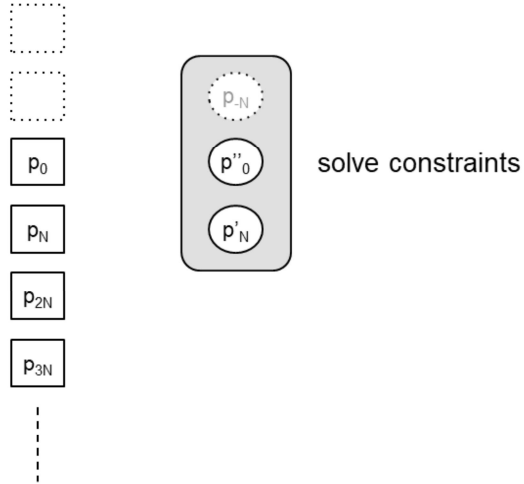
Constraint Solve



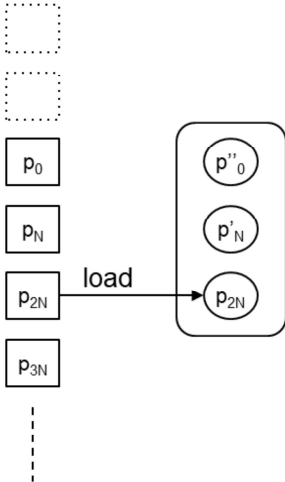
Constraint Solve



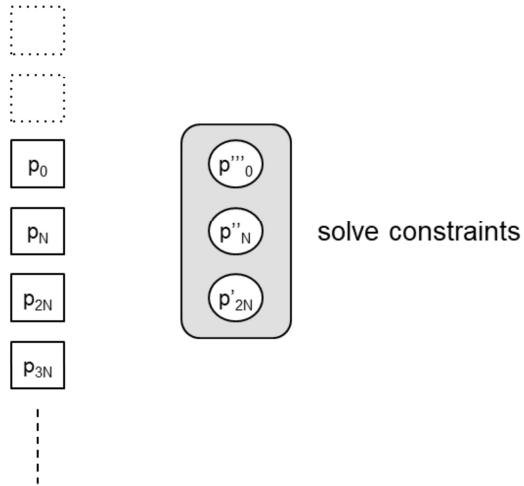
Constraint Solve



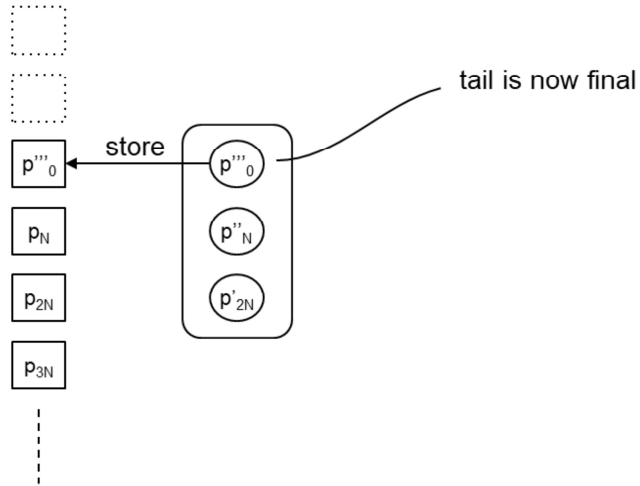
Constraint Solve

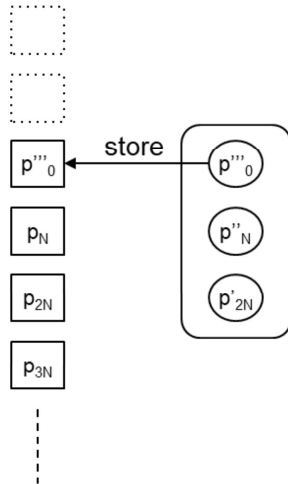


Constraint Solve

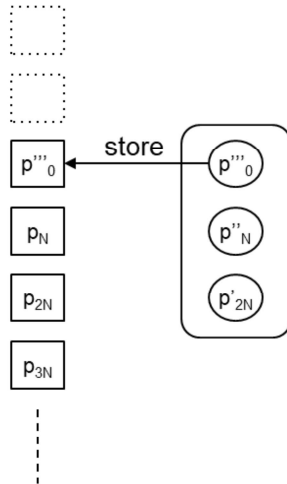


Constraint Solve

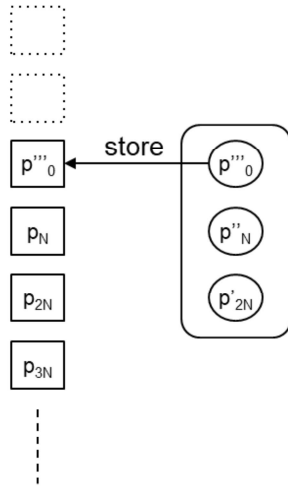




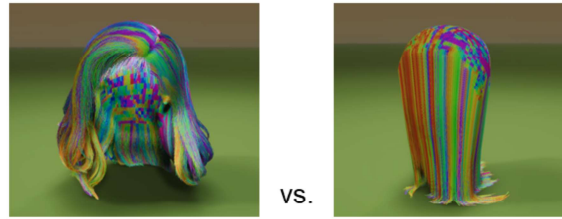
- Window size depends on active set of constraints
 - Constraints in window affect the same particle a fixed number of times down the strand
 - E.g. particle-particle distance constraint will affect each particle twice
- Compile-time variants for window size + feature flags for individual constraints
- Simpler setup => less we try to carry

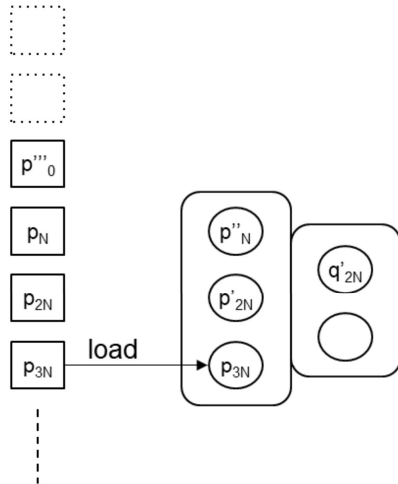


- Supported constraints
 - Boundary w/ friction [Macklin et al. 2014]
 - Distance particle-particle [Müller et al. 2006]
 - Distance particle-root [Kim et al. 2012]
 - Distance FTL [Müller et al. 2012]
 - Local bend limiter [Kelager et al. 2010]
 - Local shape [Kugelstadt and Schömer 2016]
 - Global rotation (local shape with bias)
- Order of evaluation is fixed

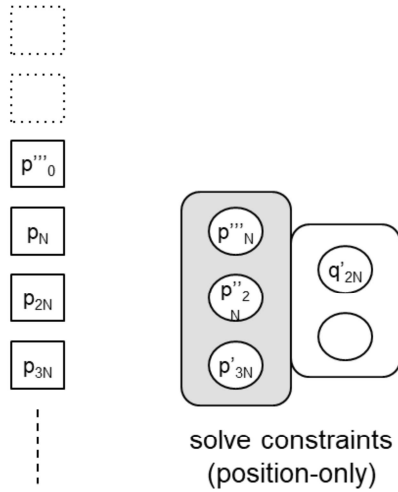


- What about orientation?
 - Fundamental to local shape
 - Resisting to load/store more data per particle
- ... local shape is rather important 😊

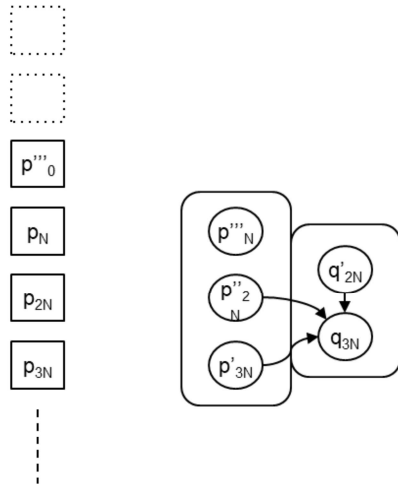




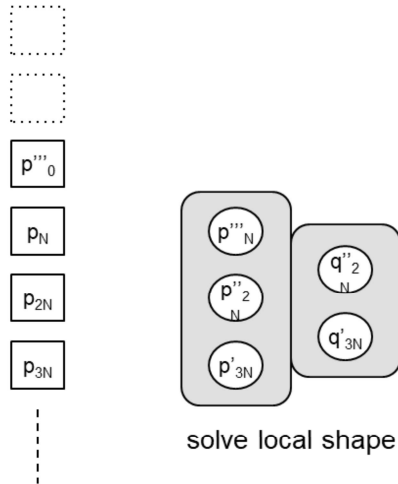
- What about orientation?
 - Fundamental to local shape
 - Resisting to load/store more data per particle
- Build frame as we move down the strand
 - Added window always has size 2



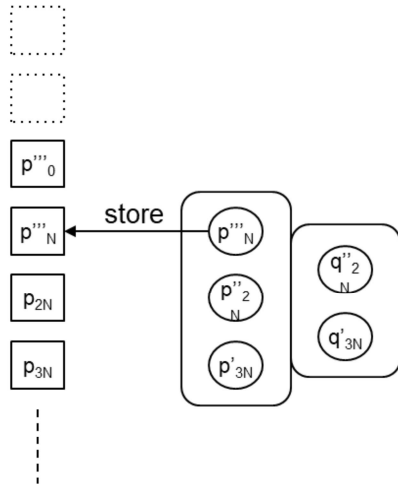
- What about orientation?
 - Fundamental to local shape
 - Resisting to load/store more data per particle
- Build frame as we move down the strand
 - Added window always has size 2
 - Position-only constraints handled first



- What about orientation?
 - Fundamental to local shape
 - Resisting to load/store more data per particle
- Build frame as we move down the strand
 - Added window always has size 2
 - Position-only constraints handled first
 - Then
 - $dq = qfromto(qmul(q'_{2N}, e^3), p'_{3N} - p'_{2N})$
 - $q_{3N} = qmul(dq, q'_{2N})$
 - ($e^3 = \text{forward}$)



- What about orientation?
 - Fundamental to local shape
 - Resisting to load/store more data per particle
- Build frame as we move down the strand
 - Added window always has size 2
 - Position-only constraints handled first
 - Then
 - $dq = qfromto(qmul(q'_{2N}, e^3), p'_{3N} - p'_{2N})$
 - $q_{3N} = qmul(dq, q'_{2N})$



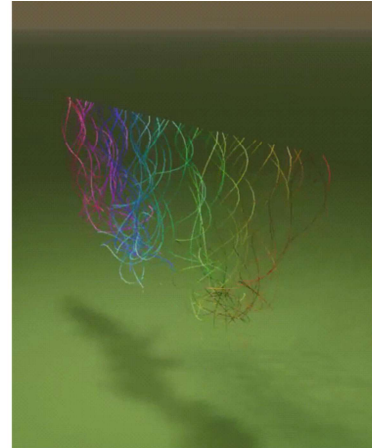
- What about orientation?
 - Fundamental to local shape
 - Resisting to load/store more data per particle

- Build frame as we move down the strand
 - Added window always has size 2
 - Position-only constraints handled first
 - Then
 - $dq = qfromto(qmul(q'_{2N}, e^3), p'_{3N} - p'_{2N})$
 - $q_{3N} = qmul(dq, q'_{2N})$
 - Still storing just position

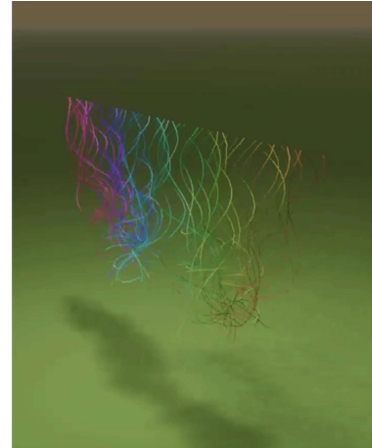
Local Shape w/ inferred material frame

- Some immediate downsides
 - Always working with rotation minimizing frame
 - Twist not preserved between iterations/steps
 - Can't twist strands to make them coil
 - Can't render e.g. textured tubes that visibly twist

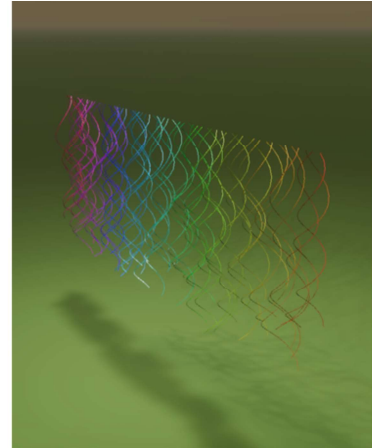
- Some immediate downsides
 - Always working with rotation minimizing frame
 - Twist not preserved between iterations/steps
 - Can't twist strands to make them coil
 - Can't render e.g. textured tubes that visibly twist
- More problematic: Stability 🤖
 - [Kugelstadt et al. 2016] suggest not solving in sequential order => would require storing (q, w)



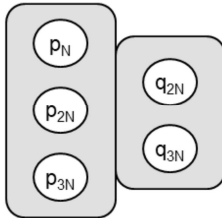
- Some immediate downsides
 - Always working with rotation minimizing frame
 - Twist not preserved between iterations/steps
 - Can't twist strands to make them coil
 - Can't render e.g. textured tubes that visibly twist
- More problematic: Stability
 - [Kugelstadt et al. 2016] suggest not solving in sequential order => would require storing (q, w)
 - Bring out the workarounds ☹
 - Angular damping, bias towards reference, composition



- Angular damping
 - Per-segment during velocity update
 - $w = \text{angular}(r, v)$
 - $w_{\text{damp}} = \text{decay}(w)$
 - $dv = \text{linear}(r, w_{\text{damp}} - w)$
- Bias towards reference
 - Internally, local shape is a composition of two constraints defined by [Kugelstadt et al. 2016]
 - $\text{bend-twist}(q_a, q_b, w_a, w_b, \dots)$
 - $\text{stretch-shear}(p_a, p_b, q, \dots)$
 - $w_a = 0.5$ makes q_a more resistant to change



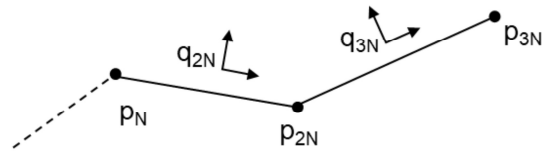
- Composition of
 - bend-twist(q_a, q_b, \dots)
 - stretch-shear(p_a, p_b, q, \dots)
- We have two modes
 - "Forward"
 - "Stitched"



solve local shape

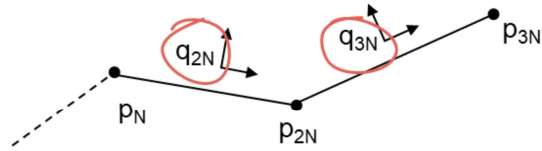
Local Shape w/ inferred material frame

- “Forward”
 - bend-twist(q_{2N}, q_{3N}, \dots)
 - stretch-shear($p_{2N}, p_{3N}, q_{3N}, \dots$)



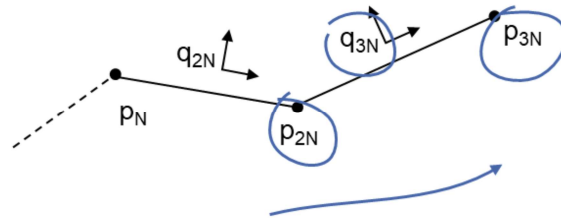
Local Shape w/ inferred material frame

- “Forward”
 - **bend-twist**(q_{2N}, q_{3N}, \dots)
 - **stretch-shear**($p_{2N}, p_{3N}, q_{3N}, \dots$)



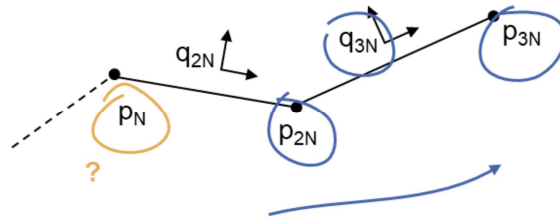
Local Shape w/ inferred material frame

- “Forward”
 - bend-twist(q_{2N}, q_{3N}, \dots)
 - stretch-shear($p_{2N}, p_{3N}, q_{3N}, \dots$)

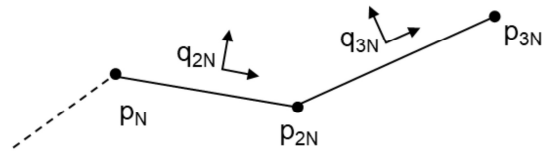


Local Shape w/ inferred material frame

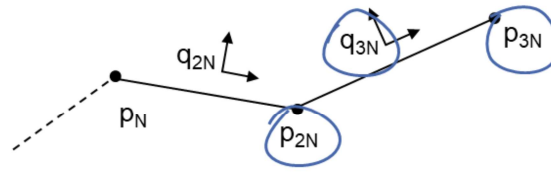
- “Forward”
 - bend-twist(q_{2N}, q_{3N}, \dots)
 - stretch-shear($p_{2N}, p_{3N}, q_{3N}, \dots$)



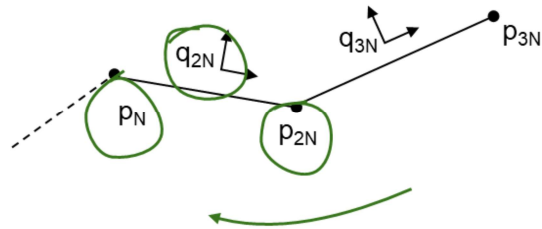
- “Stitched”
 - stretch-shear($p_{2N}, p_{3N}, q_{3N}, \dots$)
 - stretch-shear($p_N, p_{2N}, q_{2N}, \dots$)
 - bend-twist(q_{2N}, q_{3N}, \dots)



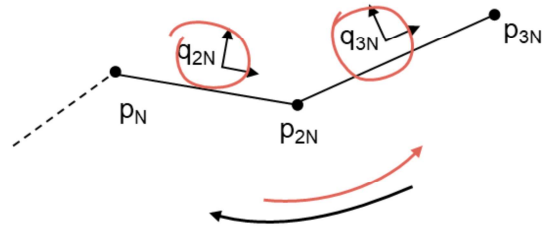
- “Stitched”
 - $\text{stretch-shear}(p_{2N}, p_{3N}, q_{3N}, \dots)$
 - $\text{stretch-shear}(p_N, p_{2N}, q_{2N}, \dots)$
 - $\text{bend-twist}(q_{2N}, q_{3N}, \dots)$



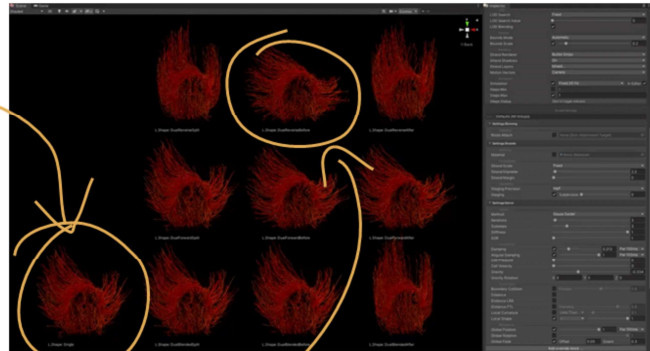
- “Stitched”
 - stretch-shear($p_{2N}, p_{3N}, q_{3N}, \dots$)
 - **stretch-shear($p_N, p_{2N}, q_{2N}, \dots$)**
 - bend-twist(q_{2N}, q_{3N}, \dots)



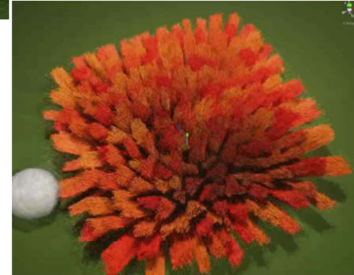
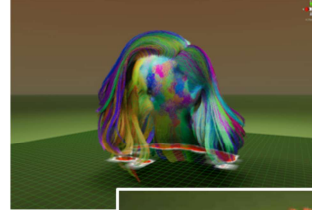
- “Stitched”
 - stretch-shear($p_{2N}, p_{3N}, q_{3N}, \dots$)
 - stretch-shear($p_N, p_{2N}, q_{2N}, \dots$)
 - **bend-twist(q_{2N}, q_{3N}, \dots)**

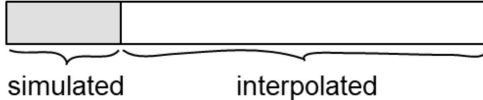


- “Forward”
 - First / immediate approach
 - Carries information only forward
 - Thought maybe we can do better
- “Stitched”
 - After parallel implementations
 - Carries information back and forth within window (hence the name)
 - Converges faster



- LOD data from just curves
 - Cluster roots, full strands, three-point simplified
 - Centroids => primary strands of single LOD
- Artistic use
 - Initially we were just simulating all the strands
 - No physical model of strands clumping together
 - Clumps easily diffusing != artistic vision
 - Simulating reduced set helps preserve clumps
 - Also cheaper
- Scalability
 - Primary strands carry volume of cluster



- Ordering 

The diagram shows a horizontal bar divided into two sections. The left section is shaded grey and labeled 'simulated' below it. The right section is white and labeled 'interpolated' below it. Brackets underneath the bar indicate the extent of each section.
- Simulation fully supports the data
 - Volume and collisions intact for lower LODs
 - Enables higher density grooms without completely breaking the budget
- See the 'Lion' demo
 - ~2M strands
 - Smart use of clustering
 - ~4ms sim. on PS5 (w/ volume)





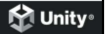
SIGGRAPH 2022
VANCOUVER+ 8-11 AUG



Physically Based Hair Shading

John Parsaie

© 2022 SIGGRAPH. ALL RIGHTS RESERVED. ADVANCES IN REAL-TIME RENDERING IN GAMES COURSE.



To conclude our presentation, I'm going to talk about some progress we've made in bringing a physically based shading model for hair to Unity.

$$S(\theta_i, \theta_r, \phi_i, \phi_r) = k_d + k_s \frac{\cos^n(\theta_r + \theta_i)}{\cos \theta_i}$$

→ **Phenomenological**

Modelled empirically rather than physically.

→ **Dual Specular Terms**

Approximation of the initial reflection (and internal reflection + transmit) of incident light.

→ **Transmission Term**

Approximation of the twice transmitted incident light.

→ **Diffuse Term**

Very coarse approximation of multiple scattering, giving hair its color (especially if low absorbing / light hair).



Prior to our work on this, Unity's HDRP already provided a hair shader available to users.

This was a non-physically based, Kajiya-Kay based model.

There are some key characteristics of this Kajiya-based model.

First to say that the model is phenomenological, meaning that it's modeled empirically, based on observation of the phenomena only *seen* in hair, rather than an approximation of the actual physics of what's really happening.

As such, our Kajiya-based BSDF models the following three phenomena:

There is first the dual specular terms.

Effectively this is approximating the initial specular lobe that forms due to reflection of incident light on a hair cuticle scale.

The second specular term is meant to approximate the specular lobe that forms from incident light that get transmitted into the hair fiber, internally reflects at the back of the cuticle, and finally transmits a second time slightly shifted from the point of entry; tinted in color due to absorption by the hair fiber cortex.

Secondly there is a special transmission term. This approximates incident light that transmits once into the hair fiber, and transmits a second time at the cuticle wall on the

opposite side.

Finally, there is the diffuse term, which ultimately is meant to coarsely approximate the multiple scattering

Especially for darker hair where multiple scattering is barely observable, you can get very, very far in terms of quality by carefully tuning this model... however, there were still some issue with achieving consistent results across lighting scenarios and a non-physically meaningful inputs to the model

→ **Intuitive Parameterization**

More physically meaningful artist inputs.

→ **Consistency**

Conform to any lighting scenario, as one would expect in a physically based renderer.

→ **Multiple Scattering**

Exhibiting the propagation of light through a volume of hair, not just a singular fiber.

To summarize our motivation to improve HDRP's hair shading model, we wanted to achieve three things:

First, a more intuitive parameterization of the model, mainly to reduce the number of color inputs required.

Of course we require it to be energy conserving and work across any lighting scenario.

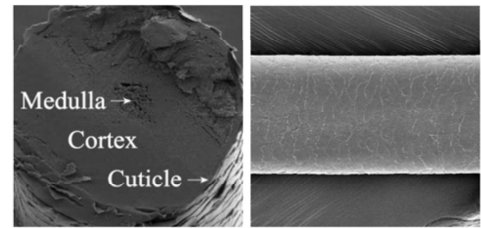
Last but not least, we would also like to approximate the advanced multiple scattering that occurs within a volume of hair fibers.

Mammal Hair Follicle Biology

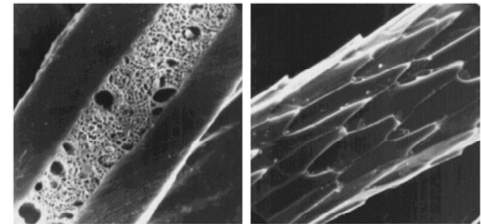
The **cuticle** is the outermost scale-like structure on the fiber. The overlapping structure has a significant effect on the reflection and refraction of incident light.

The **cortex** comprises most (~75%) of a human hair fiber interior, and is responsible for the absorption that occurs to refracted light in the fiber; giving its color.

The **medulla** is the innermost structure of the fiber and further contributes to scattering. For humans, the medulla contribution to scattering is negligible due to its small size. For animal fur, it is much larger and must be taken into account.



Human Hair Fiber Cross Section and Cuticle



Animal Fur Cross Section (left, Cougar) and Cuticle (right, Corsac Fox)

[Yan et. al. 2015]

A typical animal hair follicle is composed of three substructures: the cuticle, the cortex, and the medulla.

The Cuticle is the outermost scale-like structure on the fiber and the barrier to its interior. Cuticles are arranged similarly to roof tiles, with a slight overlap to one another and with a slight degree of inclination that effects the reflection and refraction of the incident light.

The Cortex makes up most of the fiber interior, and is most responsible for the absorption that occurs with refracted light, giving the hair its color.

The Medulla is the innermost structure of the fiber and contributes further to the scattering. For animal fur, this structure is much larger than humans and is what gives fur it's generally more diffuse appearance.

Together these three structures produce the unique scattering of incident light on hair.

Obviously it is unfeasible to explicitly geometrically model the fiber structures, and solve the scattering and absorption with a traditional microfacet model, even in a path traced setting.

We need to look to a way that simplifies this structure and the way it reflects, refracts, and absorbs incident light.

Marschner's Factorization

→ BCSDF

- Bidirectional Curve Scattering Distribution Function
- Decompose the fiber scattering analysis into the product of two 1D profiles.

→ Longitudinal Scattering

- Models the scattering along the length of the fiber.

→ Azimuthal Scattering

- Models the scattering along the width of the fiber (parameterized by h in the diagram).

$$\begin{aligned} \cdot(\omega_r) &= \int L_i(\omega_i) S(\omega_i, \omega_r) \cos \theta_i d\omega_i, \\ S(\theta_i, \theta_r, \phi_i, \phi_r) &= \sum_p S_p(\theta_i, \theta_r, \phi_i, \phi_r) / \cos^2 \theta_d \\ &= \sum_p M_p(\theta_h) \cdot N_p(\phi; \eta') / \cos^2 \theta_d. \end{aligned}$$

(Left) Longitudinal-Azimuthal Parameterization (Middle)

[Yan et al. 2015]

So if you have looked into implemented physically based hair shading before, then diagrams like the one on the right are more than likely burned into you mind. But to ensure everyone has full context, let me provide a quick recap.

Hair reflectance models consider hair fibers as cylinders, and use a BCSDF (Bidirectional Curve Scattering Distribution Function) to represent the scattering that occurs on and within a fiber.

The Kajiya-Kay model for example is a BCSDF, and it treats hair as opaque solid cylinders.

There is already nearly two decades of research on the topic of physically based hair shading, so we have a lot to work from.

Nearly all of that research rests on Marschner's factorization, which essentially showed that it's actually possible to separate the scattering analysis for each path in a fiber into a product of two 1D profiles, which you can see in the diagram on the right.

The longitudinal scattering is an analysis of the scattering along the length of the fiber and the azimuthal scattering is an analysis of the scattering along the width of the fiber.

You will also notice in the diagram that there is really no consideration for the medulla structure. This is because since we are targeting human hair for now, and since the medulla is very small, it can be safely factored out of the model.

This leaves us with a model that treats hair fibers as glass-like dielectric cylinders.

Path Traced Reference

Factored Reflectance Model

$$S(\theta_i, \theta_r, \phi) = \sum_{p=0}^{\infty} S_p(\theta_i, \theta_r, \phi)$$

Component/Path Scattering Function

$$S_p(\theta_i, \theta_r, \phi) = M_p(\theta_i, \theta_r) N_p(\theta_i, \theta_r, \phi)$$

Energy Conserving Longitudinal Scattering

$$M_p(v, \theta_i, \theta_r) = \frac{\operatorname{csch}(1/v)}{2v} e^{\frac{\sin(-\theta_i)\sin\theta_r}{v}} I_0\left[\frac{\cos(-\theta_i)\cos\theta_r}{v}\right]$$

Roughened Near-Field Azimuthal Scattering

$$N_p(\phi, h) = A_p(h) D_p(\phi - \Phi(p, h))$$

Auxiliary

Fiber Attenuation

$$A(p, h) = (1 - f)^2 f^{p-1} T(\mu_a', h)^p$$

Logistic Distribution

$$D(\phi) = I_g(\phi, s; -\pi, \pi).$$

Azimuthal Change

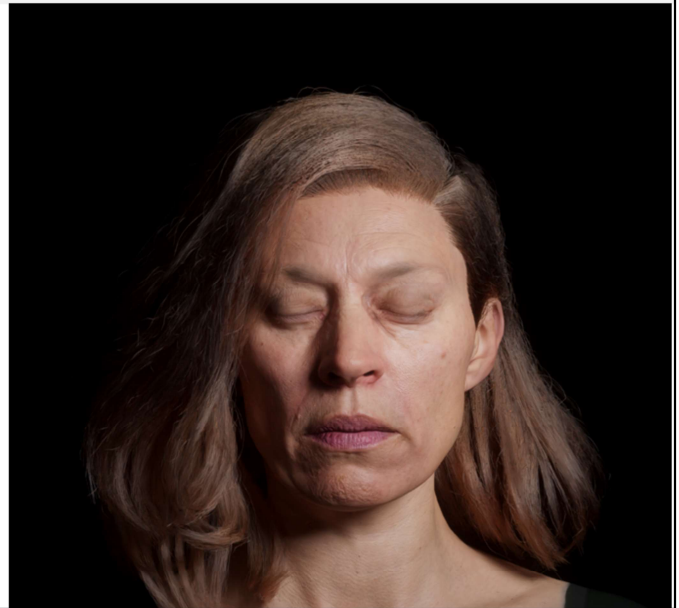
$$\Phi(p, h) = 2p\gamma_l - 2\gamma_i + p\pi$$

Absorption

$$T(\mu_a, h) = \exp(-2\mu_a(1 + \cos(2\gamma_l)))$$

Modified Bessel Function of the First Kind

$$I_0(x) \approx \sum_{i=0}^{10} \frac{x^{2i}}{4^i (i!)^2}$$



So now let's discuss a little bit about implementation.

To begin, we decided it was crucial to begin with a path traced reference before attempting to approximate anything.

This way, we could make well informed optimization decisions. As well as have a good understanding of the usual visual vs. performance tradeoffs.

The equations in this slide are for future reference when these slides are shared online.

If there was more time to speak I would go in greater detail here, but for now just know that these are the building blocks for our path-traced model, and what we will eventually try to approximate.

→ Can't solve for an infinite series of paths. Let's simplify to the first three.

$$S(\theta_i, \theta_r, \phi) = \sum_{p=0}^{\infty} S_p(\theta_i, \theta_r, \phi)$$

→ The longitudinal scattering function is pretty costly.

$$S_p(\theta_i, \theta_r, \phi) = M_p(\theta_i, \theta_r) N_p(\theta_i, \theta_r, \phi)$$

→ Evaluating the near field azimuthal scattering (even once) is costly and would result in a lot of aliasing. Numerically integrating over the fiber width is also prohibitive in cost.

$$M_p(\nu, \theta_i, \theta_r) = \frac{\operatorname{csch}(1/\nu)}{2\nu} e^{\frac{\sin(-\theta_i) \sin \theta_r}{\nu}} I_0 \left[\frac{\cos(-\theta_i) \cos \theta_r}{\nu} \right]$$

$$N_p(\phi, h) = A_p(h) D_p(\phi - \Phi(p, h))$$

Speaking of approximation, let's talk about that now.

Here is a closer look at the primary equations from our path traced model that we need to approximate.

Some initial observations:

- Immediately we see that it is not feasible to solve for an infinite series of paths. So we will compromise on the loss of about 15% of the energy and only solve for the first three paths.
- We'll also see that the energy conserving longitudinal scattering is pretty costly compared to Marschner's original gaussian.
- Finally we will see that the near field azimuthal scattering is almost unusable, since before we were relying on the monte carlo integration of the path tracer to integrate this function over the fiber width. So only sampling it once would result in a lot of aliasing and inaccurate results. Additionally, we cannot simply numerically integrate the function for a far-field result because this would also be too expensive.

We already decided to simplify to the first three paths. I will explain next how we simplified the longitudinal scattering and azimuthal scattering.

$$M_p(v, \theta_i, \theta_r) = \frac{\operatorname{csch}(1/v)}{2v} e^{\frac{\sin(-\theta_i) \sin \theta_r}{v}} I_0 \left[\frac{\cos(-\theta_i) \cos \theta_r}{v} \right]$$

$$M_p = g(\beta_p; \theta_h - \alpha_p)$$

$$g(\beta; \theta) = \frac{e^{-\theta^2/(2\beta^2)}}{\sqrt{2\pi}\beta}$$

So for longitudinal scattering, I have this slide simply for completeness' sake, since we do what others in the past have done already.

For all three paths, we simply defer to the original Marschner gaussian in place of the energy conserving one.

The gaussian takes the standard deviation representing the fiber roughness, and it is a function of the half angle between the longitudinal light and camera vector.

The alpha term represents the cuticle angle in radians, and this is how we factor in the shifting that occurs due to the reflection and refraction on the cuticles.

$$N_p(\phi, h) = \underbrace{A_p(h)}_{\text{blue}} \underbrace{D_p(\phi - \Phi(p, h))}_{\text{orange}}$$

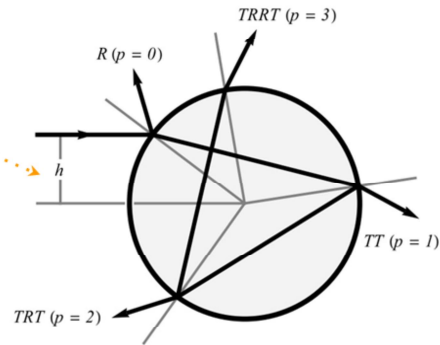
For the azimuthal scattering we have to be a little more creative.

We take the approach of approximating the azimuthal scattering to be the product of the approximated attenuation and approximated distribution for each path.

First I will talk about what we did for the distribution function underlined in orange, and then the attenuation function underlined in blue.

A reminder that attenuation accounts for the fresnel and absorption that occurs at each event, while the distribution determines the radial angular distribution.

$$N_p(\phi, h) = \underbrace{A_p(h)}_{\text{blue}} \underbrace{D_p(\phi - \Phi(p, h))}_{\text{orange}}$$



[d'Eon et. al 2013]

And an extra reminder to explain the H term.


If you look back at our diagram explaining the analysis of the azimuthal scattering in the normal plane, you can visually see that H represents the offset of the incident light from the fiber.

Since we analyze this on the unit circle, H is in the range of -1 to +1.

The near field model works by computing the exact H value based on the normal of fiber and the incident angle. The far field model works by integrating for H over the entire fiber width.

- Pre-integrate the far-field distribution and store it in a 3D LUT.
- Parameterized by **phi**, **cosine theta d**, and **radial smoothness**.
- Computes the distributions for all three paths in one sample.

$$\text{LUT}(\phi, \cos \theta_d, \beta_N)$$


$$\int_{-1}^{+1} D_p(\phi - \Phi(p, h)) dh$$

We would really like to preserve the radial smoothness parameter as it gives artists a lot of flexibility for species differentiation even though we still mostly try to focus on human hair.

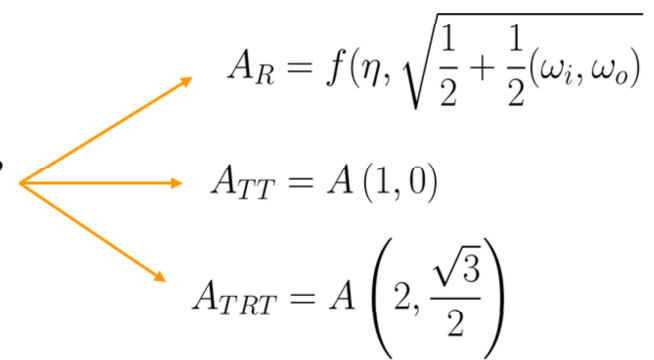
Because of this, we decided to pre-integrate the distribution over the fiber width and store it in a 3D look up table, this basically gives us the far field result at each sample.

It's parameterized by phi, the azimuthal angle around the fiber

The cosine of the angle theta d, which is the longitudinal difference angle between the light and camera vector.

And finally beta N, which represents the azimuthal roughness (to users, radial smoothness).

Since the distribution is just a scalar for each path, and the LUT sample is path independent, we can get the distributions for all three paths with one sample.

$$A(p, h) = (1 - f)^2 f^{p-1} T(\mu_a', h)^p$$

$$A_R = f(\eta, \sqrt{\frac{1}{2} + \frac{1}{2}(\omega_i, \omega_o)})$$
$$A_{TT} = A(1, 0)$$
$$A_{TRT} = A\left(2, \frac{\sqrt{3}}{2}\right)$$

Now on to the approximation for the attenuation.

The R path we do not need to do anything special as no absorption occurs.

For TT and TRT we took an approach as demonstrated by Epic and Frostbite which basically tries to select the dominant H value for each path.

For TT this ended up being 0, which makes sense since the forward scattering is observed to be most dominant when incident light strikes a fiber dead-on.

For TRT we select the H term of $\sqrt{3} / 2$ for similar reasoning.

Multiple Scattering

We are not just shading one hair in isolate, we are shading one amongst tens to hundreds of thousands nearby.

Incident light that transmits out of a fiber without being fully absorbed and arrives at a new fiber.

Most obvious / important for lighter colored (lower absorbing) hair.



One more thing of note, It's also not enough to consider this problem in isolate of a single hair fiber, for good results we need to factor in the fact that we will be shading an entire volume of hair.

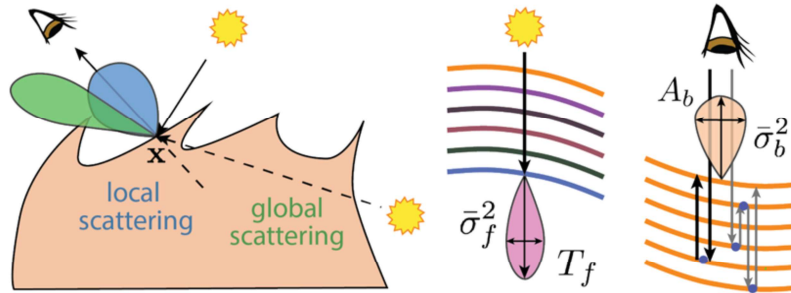
The reason for this is most easily seen in a blonde (or low absorbing) head of hair.

Since the absorption is low for the fiber, the light is able to scatter further into the volume of strands.

This is what gives lighter colored hair its soft and volumetric appearance.

So, we must take this into account as well.

$$\Psi(x, \omega_d, \omega_i) = \Psi^G(x, \omega_d, \omega_i) (1 + \Psi^L(x, \omega_d, \omega_i))$$



[Yan et. al. 2017]

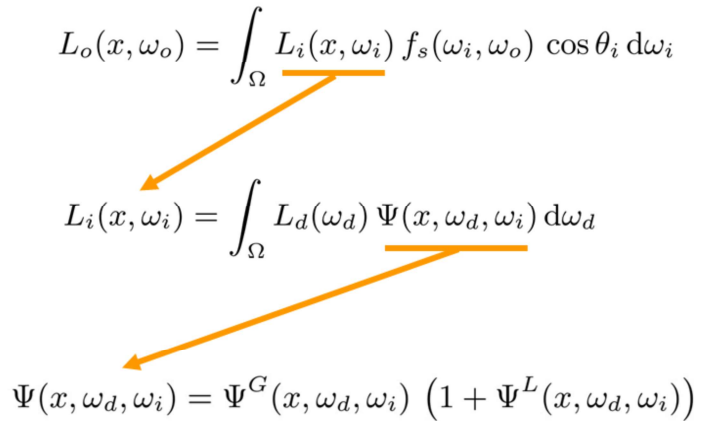
The main idea of the dual scattering method is to approximate the multiple scattering function with a Global Multiple Scattering (Psi G) and Local Multiple Scattering component (Psi L).

The global multiple scattering function computes the irradiance arriving at the neighborhood of the shading point inside the hair volume.

The local multiple scattering function approximates the multiple scattering of this irradiance within the neighborhood of the shading point.

So basically, the multiple scattering function is the sum of the global multiple scattering and the global multiple scattering that undergoes further local multiple scattering.

- Need to better define the incoming radiance
- Use the Dual Scattering Approximation

$$L_o(x, \omega_o) = \int_{\Omega} L_i(x, \omega_i) f_s(\omega_i, \omega_o) \cos \theta_i d\omega_i$$
$$L_i(x, \omega_i) = \int_{\Omega} L_d(\omega_d) \Psi(x, \omega_d, \omega_i) d\omega_d$$
$$\Psi(x, \omega_d, \omega_i) = \Psi^G(x, \omega_d, \omega_i) (1 + \Psi^L(x, \omega_d, \omega_i))$$


To build an intuition for what causes multiple scattering in hair let's revisit the original integral that gives us the outgoing radiance in a certain direction from a given point.

The top integral shows that to compute the outgoing radiance we integrate over the sphere, the product of our hair BSDF and the incident radiance.

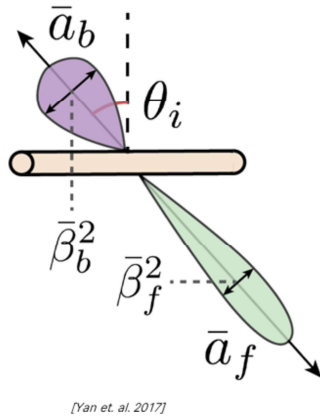
We already know about our fiber scattering model, so it's the incident radiance that we care about.

When observing the incoming radiance, we can decompose it into two contributions, the first (L_d) is the radiance incoming from outside of the hair volume.

The second is the radiance that is scattered inside of the hair volume and finally arriving at the shading point, we can call this the multiple scattering function.

In a path traced setting it's extremely easy to get this information since this integral is solved directly via monte carlo, but in a rasterized setting we really have to approximate.

We use the dual scattering method to approximate this multiple scattering function which can be seen in the bottom equation.



[Yan et. al. 2017]

$$\bar{a}_{f|b}(\theta_i) = \frac{1}{\pi} \iint_{\Omega_{f|b}} \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} S(\theta_i, \phi_i, \omega_r) \cos(\theta_r) d\phi_i d\omega_r$$

Both the global and local multiple scattering are dependant on the average forward and backward scattering intensity.


We simplify the more complicated scattering lobes into a generalized singular forward lobe and singular backward lobe, this is best explained visually in the diagram.

We compute this by solving this integral on the front and back hemisphere. Note how we directly re-use our BCSDf.

We do this as a pre-integration step and store the result it in a 3D LUT.

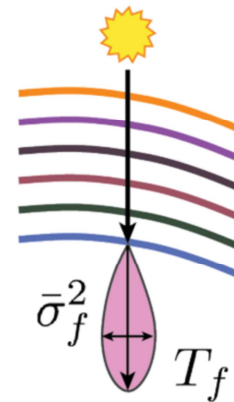
Approximating the Global Scattering Function

Global Multiple Scattering $\Psi^G(x, \omega_d, \omega_i) \approx T_f(x, \omega_d) S_f(x, \omega_d, \omega_i)$

Transmittance  $T_f(x, \omega_d) = d_f(x, \omega_d) \prod_{k=1}^n \bar{a}_f(\theta_d^k)$

Forward Scattering Spread $S_f(x, \omega_d, \omega_i) = \frac{\bar{s}_f(\phi_d, \phi_i)}{\cos \theta_d} g(\theta_d + \theta_i, \bar{\sigma}_f^2(x, \omega_d))$

Total Variance  $\bar{\sigma}_f^2(x, \omega_d) = \sum_{k=1}^n \bar{\beta}_f^2(\theta_d^k)$



[Yan et. al. 2017]

The local scattering can be computed analytically and I won't mention anything else about that, but the global scattering function is a very intensive aspect of the dual scattering approximation.

The core of this is because we need to compute the average attenuation and variance for each hair fiber intersection that occurs between the shading point and the light, best explained again by the diagram and equations.

The equations listed here are a decomposition of the global multiple scattering equation. Upon closer inspection, we can see that the transmittance and total variance calculations are the primary culprits for the cost of this routine. N here represents the number of strands between the shading position and the light.

We basically need to compute those equations at every fiber intersection. How do we do it?

Traditionally this is solved by using deep opacity maps for example at Frostbite, which are basically a more robust shadow map algorithm, but we didn't have time to sort this out on the CPU side and to be honest, it doesn't seem like a very great option to have one of these maps for every light that you want to contribute to multiple scattering.

Almost... but what about n ?

$$T_f(x, \omega_d) = d_f(x, \omega_d) \prod_{k=1}^n \bar{a}_f(\theta_d^k)$$

$$T_f(x, \omega_d) \approx d_f \bar{a}_f(\theta_d)^n$$

$$\bar{\sigma}_f^2(x, \omega_d) = \sum_{k=1}^n \bar{\beta}_f^2(\theta_d^k)$$

$$\bar{\sigma}_f^2(x, \omega_d) \approx \bar{\beta}_f^2(\theta_d) \times n$$

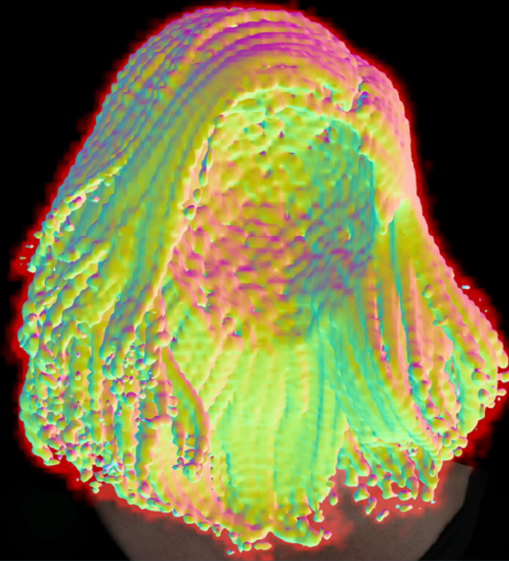
Ignoring the fact that we need to somehow get the number of strands between the shading point and the light, we can at least make this initial simplification proposed by Disney.

Basically what this does is make the assumption that all strands that exist between the shading point and the light have the exact same orientation.

This allows us to move the fiber-dependent information out of the sum and product, effectively decoupling them from the strand count.

So we have at the very least isolated this problem into computing the strand count in the shadow ray direction as fast as possible.

Leverage the volumetric artifacts produced by the simulation that occurred earlier in the frame.



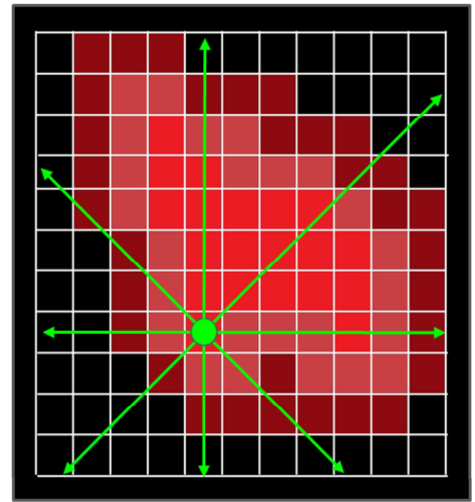
To compute N , we can benefit from the voxelized hair structure that is produced by the hair simulation that Lasse presented in the previous section of this talk.

The voxelized structure already produces numerous useful volumetric data, like velocity, density, pressure, and so on.

Pictured above is a quick visualization of the voxelized structure of the hair and some of its data, drawing the isosurface and strand density.

Approximating the Global Scattering Function

- Trace against the density volume over the sphere of directions.
- At each step, convert the voxel density into strand count, and aggregate to the trace result.
- Finally, project all of it down into an L1 band spherical harmonic and store in a new volume buffer.
- Sample the closest voxel to the shading position on the main pass and submit the L1 coefficients to the BSDFData.
- Quickly decode the spherical harmonic for any light type and use the multiple scattering algorithm as normal.



2D Voxel Grid Cross Section. Cell Redness = Density

Here's what we do.

We begin by knowing that the density data has already been calculated for each voxel.

The density at each voxel represents the fractional occupancy of strands to the total volume of the voxel.

It is a term that we can directly derive an approximate strand count per-voxel from.

So for each voxel, we trace against the density volume over the sphere of directions until we hit the volume extents.

For every step, we convert the density into strand count, and aggregate it to the current trace result.

Then we project all of this down into an L1 Band spherical harmonic and store it in a new volume buffer.

During shading, sample from the closest voxel to the shading position, and submit the L1 harmonic to the BSDFData.

During the lightloop evaluation, we can quickly decode the coefficients and retrieve the approximate strand count for any light type, including area lights.

This allows us to compute the multiple scattering for both analytic and area lights whether or not those lights cast shadows.

Tile-Based Line Software Rasterization



Finally, I wanted to quickly explain a topic very unrelated to shading, but nonetheless extremely important for rendering hair.

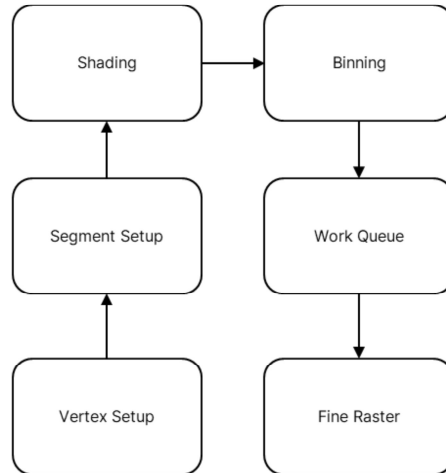
And that topic is software rasterization for hair strands.

We were deeply inspired by Frostbite's demonstration of a compute-based rasterizer to solve the infamous problems of hair strands rendering, mainly being anti-aliasing and fragment composition for proper transparency, two things which the traditional graphics pipeline struggles with in terms of both image quality and performance for strand geometry.

I will defer you to their excellent 2020 talk on this subject, which goes into far greater detail than I can now due to time.

For now, I will explain to you some of our approach, which is similar in spirit, but with its differences.

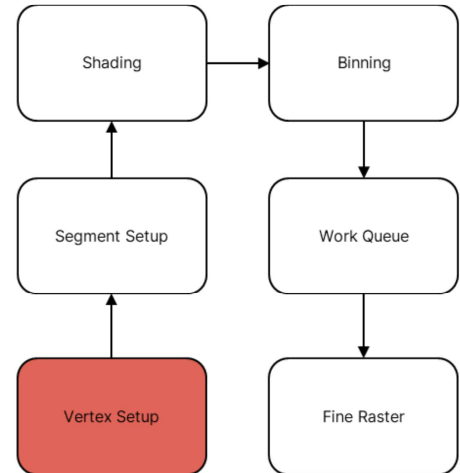
Tile-Based Line Software Rasterization



Here is a quick overview of the stages required to complete a draw call of some line topology in our software rasterizer.

Vertex Setup

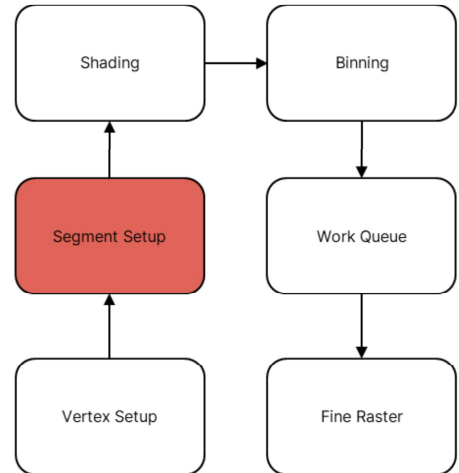
- Creates the post-transform vertex cache.
- Dispatched in compute.
- Requires you to generate your material graph for compute.
- Could do vertex stage UAV output, but that's not really supported on modern consoles.
- Very simple one-to-one output.



First the vertex setup. This is a very simple kernel, that scales in complexity with the material graph. We dispatch this in compute, rather than in the normal graphics pipe. It is 1 lane per Vertex and outputs a post transform vertex cache.

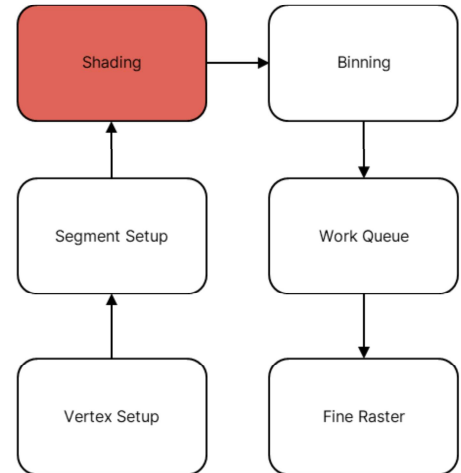
Segment Setup

- Determine segment visibility.
- Viewport clipping.
- Cull segments against depth prepass result from earlier in the frame.
- Can also cull segments more aggressively by using the volumetric data from the simulation.
- Output compacted stream of visible segment records.



Next is the segment setup. This is where we bind the index buffer and determine segment connectivity and visibility. Using the result from the previous vertex setup stage, we perform viewport clipping, and other visibility culling tests. Here we output a compacted stream visible segments that may proceed to the latter stages.

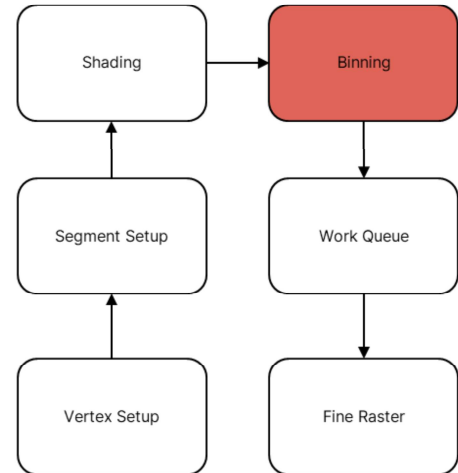
- Render shading samples into an atlas.
- Since the BCSDf is a far-field model, can shade one dimensionally.
- Currently just shade vertices for segments that pass the culling tests.
- Interpolate the shading samples later in the fine raster stage.



Up next we have the shading pass. Here we render shading samples into an atlas for every visible segment. Since our hair BCSDf is a far-field model, we can get away with shading 1-dimensionally.

Right now we have one shading sample per visible vertex, but there's room for improvement in the future. These shading samples are interpolated later, in the fine raster stage.

- Indirect dispatch from visible segment counter.
- One segment per lane.
- Compute segment bounding box.
- Segment-Tile intersection tests.
- 8x8 pixel tiles.
- Intersections emit a tile record into a big list in global memory, iterate per-tile counters.



Next is the binning stage. This stage is indirectly dispatched based on our visible segment counter. Every lane is assigned a segment.

The segment's tile space bounding box is assigned computed, and we then perform a segment intersection test for each tile in the bounding box.

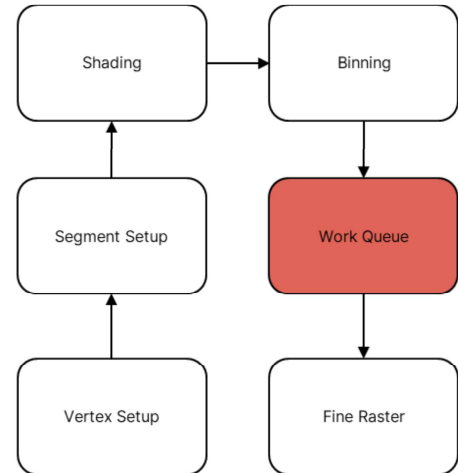
Tile's have a dimension of 8 by 8 pixels.

For passing intersections, we emit a recording of the intersection into global memory.

These recordings are coordinated through wave intrinsics and local atomics.

The output of this stage represents a long list of work that needs to be organized / rearranged before we can begin resolving per-fragment coverage and color.

- Re-organize the big list of work into something manageable for the fine rasterizer stage.
- Segment Keys (MSB 8-bit Depth, LSB 24-bit Index)
- Contiguous in memory based on resident tile.
- Secondary count / offset buffer for processing segment keys.



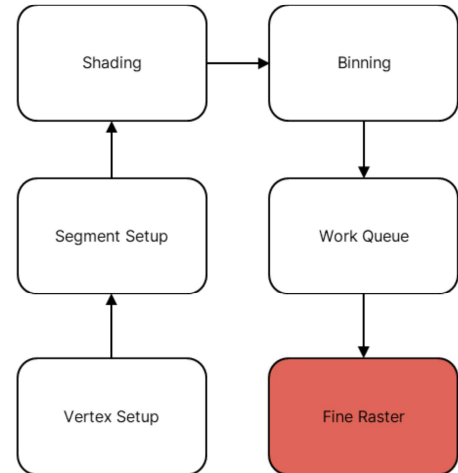
So now it's time to organize the data. In the work queue stage, the goal is to prepare the segment-tile intersection records so that we have good work distribution on the subsequent fine raster stage.

Rather than getting too into the weeds on what we do to organize this list of work, I will just say that it ultimately produces two important lists:

The first list is just a massive list of segment keys, with a \log_2 encoded depth in the 8 Most significant bits, and the actual segment index in the 24 least significant bits. These keys are organized such that all segments in the same tile are contiguous in memory.

The second list is a list that allows us to coherently look up into that first segment index buffer, providing an offset and count into that buffer.

- Persistent thread e.g. while(there is work)
- Block size of 64 threads (8x8 tile).
- Coordinate tile selection via global atomic.
- Once tile is selected for a block, load the offset and count into segment key buffer.
- Each thread loads batches of segment keys into LDS.
- Sort the 8 MSB of the keys.



Last but most definitely not least is the fine raster stage. In this final stage it is our goal to produce and compose anti-aliased fragments in a correctly sorted order as fast as possible.

This kernel follows a persistent thread style, meaning that the size of our dispatch will never be with respect to the amount of work that needs to get done, but rather how much resources there physically is in the hardware, and we will manage the work distribution ourselves.

The benefit of doing this is that we can process our work queue in ways that we can control, ultimately resulting in better performance, rather than if the work was scheduled more randomly.

Each thread block has a size of 64 threads and they will run until the tile queue is empty.

Coordination between thread block happens once via a global atomic where the next tile index is popped from the queue.

Then, we load that tile's segment offset and count. Each thread is responsible for loading batches of segment keys into LDS.

We then perform an LDS-based sort on the 8 most significant bits of the segment keys, producing a re-arranged LDS list of segments in the tile from front to back order.

[CLICK]

Now that all the data is right where it needs to be, let's rasterize.

We process the sorted segments from front to back in batches of 32. We load these batches in parallel, pulling the actual segment record from memory with the segment index that was encoded in the sorted key.

Once we have our segment records in local memory, we compute a coverage mask that is stored in an 8x8 uint tile in local memory.

Threads coordinate to render out 8 points for each segment into the coverage mask, doing this for all 32 segments in the batch.

Coverage is easily determined by transforming the NDC position onto the LDS tile.

Since the segments are already sorted, ordering of the coverage mask is easily preserved by setting the Nth bit in the coverage mask based on the Nth segment's index in the batch. Coverage mask samples are composited with an atomic OR.

[CLICK]

Once we have finished generating the coverage mask, we dilate the mask using an atomic OR. The dilation makes the mask wide enough for when we produce anti-aliased fragments.

Finally, each lane in the thread group grabs a uint from the mask. This 32-bit element from the mask represents a list of ordered segments for that pixel to produce fragments for.

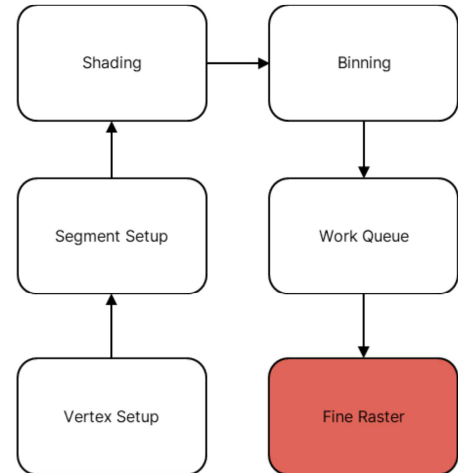
This is only place in the kernel where the threads diverges, selecting their next segment index to process with the `firstbitlow()` intrinsic.

After producing a fragment, the result is blended, the bit in the per-lane work queue is cleared, and we repeat this process again as long as the per lane queue is non-zero.

This entire batched raster process is repeated until the tile is deemed to be opaque, something we ballot for with a wave intrinsic that checks up on the minimum opacity in across the tile. If it's around 1, we are done rasterizing that tile.

Once we are done rasterizing a tile to the internal color, depth, and motion vector targets, we pick a new one and restart the process, otherwise we exit the loop and are done with the rasterization process.

- Now we have a list of segments in tile sorted from front to back order.
- Process sorted segments in batches of 32.
- Extract segment index from the LSB.
- Compute an 8x8 UINT Coverage Mask in LDS.
- Lanes coordinate to render 8 points-per-segment into mask.
- Segments already sorted, just set the Nth bit in coverage mask based on Nth segment index in batch.
- Composite with Atomic OR.



Last but most definitely not least is the fine raster stage. In this final stage it is our goal to produce and compose anti-aliased fragments in a correctly sorted order as fast as possible.

This kernel follows a persistent thread style, meaning that the size of our dispatch will never be with respect to the amount of work that needs to get done, but rather how much resources there physically is in the hardware, and we will manage the work distribution ourselves.

The benefit of doing this is that we can process our work queue in ways that we can control, ultimately resulting in better performance, rather than if the work was scheduled more randomly.

Each thread block has a size of 64 threads and they will run until the tile queue is empty.

Coordination between thread block happens once via a global atomic where the next tile index is popped from the queue.

Then, we load that tile's segment offset and count. Each thread is responsible for loading batches of segment keys into LDS.

We then perform an LDS-based sort on the 8 most significant bits of the segment keys, producing a re-arranged LDS list of segments in the tile from front to back order.

[CLICK]

Now that all the data is right where it needs to be, let's rasterize.

We process the sorted segments from front to back in batches of 32. We load these batches in parallel, pulling the actual segment record from memory with the segment index that was encoded in the sorted key.

Once we have our segment records in local memory, we compute a coverage mask that is stored in an 8x8 uint tile in local memory.

Threads coordinate to render out 8 points for each segment into the coverage mask, doing this for all 32 segments in the batch.

Coverage is easily determined by transforming the NDC position onto the LDS tile.

Since the segments are already sorted, ordering of the coverage mask is easily preserved by setting the Nth bit in the coverage mask based on the Nth segment's index in the batch. Coverage mask samples are composited with an atomic OR.

[CLICK]

Once we have finished generating the coverage mask, we dilate the mask using an atomic OR. The dilation makes the mask wide enough for when we produce anti-aliased fragments.

Finally, each lane in the thread group grabs a uint from the mask. This 32-bit element from the mask represents a list of ordered segments for that pixel to produce fragments for.

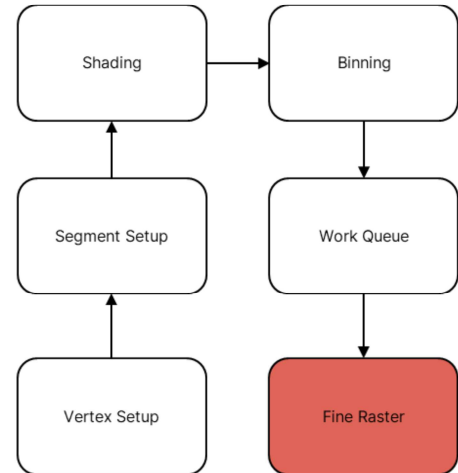
This is only place in the kernel where the threads diverges, selecting their next segment index to process with the `firstbitlow()` intrinsic.

After producing a fragment, the result is blended, the bit in the per-lane work queue is cleared, and we repeat this process again as long as the per lane queue is non-zero.

This entire batched raster process is repeated until the tile is deemed to be opaque, something we ballot for with a wave intrinsic that checks up on the minimum opacity in across the tile. If it's around 1, we are done rasterizing that tile.

Once we are done rasterizing a tile to the internal color, depth, and motion vector targets, we pick a new one and restart the process, otherwise we exit the loop and are done with the rasterization process.

- Dilate coverage mask with another set of Atomic OR.
- Each lane grabs a UINT from the tile mask.
- The 32-bit mask represent a list a per-lane work queue or ordered segments.
- Lanes diverge here, selecting the segment to compute a fragment with **firstbitlow**.
- Continue until per-lane work queue is zero.
- Check on tile opacity with WaveActiveMin. If ~1, we're done.
- Repeat.



Last but most definitely not least is the fine raster stage. In this final stage it is our goal to produce and compose anti-aliased fragments in a correctly sorted order as fast as possible.

This kernel follows a persistent thread style, meaning that the size of our dispatch will never be with respect to the amount of work that needs to get done, but rather how much resources there physically is in the hardware, and we will manage the work distribution ourselves.

The benefit of doing this is that we can process our work queue in ways that we can control, ultimately resulting in better performance, rather than if the work was scheduled more randomly.

Each thread block has a size of 64 threads and they will run until the tile queue is empty.

Coordination between thread block happens once via a global atomic where the next tile index is popped from the queue.

Then, we load that tile's segment offset and count. Each thread is responsible for loading batches of segment keys into LDS.

We then perform an LDS-based sort on the 8 most significant bits of the segment keys, producing a re-arranged LDS list of segments in the tile from front to back order.

[CLICK]

Now that all the data is right where it needs to be, let's rasterize.

We process the sorted segments from front to back in batches of 32. We load these batches in parallel, pulling the actual segment record from memory with the segment index that was encoded in the sorted key.

Once we have our segment records in local memory, we compute a coverage mask that is stored in an 8x8 uint tile in local memory.

Threads coordinate to render out 8 points for each segment into the coverage mask, doing this for all 32 segments in the batch.

Coverage is easily determined by transforming the NDC position onto the LDS tile.

Since the segments are already sorted, ordering of the coverage mask is easily preserved by setting the Nth bit in the coverage mask based on the Nth segment's index in the batch. Coverage mask samples are composited with an atomic OR.

[CLICK]

Once we have finished generating the coverage mask, we dilate the mask using an atomic OR. The dilation makes the mask wide enough for when we produce anti-aliased fragments.

Finally, each lane in the thread group grabs a uint from the mask. This 32-bit element from the mask represents a list of ordered segments for that pixel to produce fragments for.

This is only place in the kernel where the threads diverges, selecting their next segment index to process with the `firstbitlow()` intrinsic.

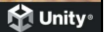
After producing a fragment, the result is blended, the bit in the per-lane work queue is cleared, and we repeat this process again as long as the per lane queue is zero.

This entire batched raster process is repeated until the tile is deemed to be opaque, something we ballot for with a wave intrinsic that checks up on the minimum opacity in across the tile. If it's around 1, we are done rasterizing that tile.

Once we are done rasterizing a tile to the internal color, depth, and motion vector targets, we pick a new one and restart the process, otherwise we exit the loop and are done with the rasterization process.



© 2022 SIGGRAPH. ALL RIGHTS RESERVED. ADVANCES IN REAL-TIME RENDERING IN GAMES COURSE.



Before I go, I will leave you with some performance numbers captured on PS5, from a demo we released yesterday, that uses all of the features we just presented.

- Attachments: 1.4ms
- Hair Sim: 3.5ms
- Strand Count Probe for Lion Mane: 0.7ms
- Shading: 3.36ms shading pass (collectively) on PS5
- Fine Raster: 10ms

References: Probe Lighting

- [Silvennoinen and Timonen 2015] A. Silvennoinen, V. Timonen, Multi-Scale Global Illumination in Quantum Break, SIGGRAPH 2015.
- [Garcia et al. 2020] K. Garcia, A. Lindqvist, A. Brink, "Hustle by Day, Risk it all at Night": The Lighting of Need for Speed Heat in Frostbite, SIGGRAPH 2020 talks
- [Hobson19] J. Hobson, The Indirect Lighting Pipeline of God of War, GDC 2019
- [O'Donnell18] Y. O'Donnell, Precomputed Global Illumination in Frostbite, GDC 2018
- [McGuire19] M. McGuire, Ray-Traced Irradiance Fields, GDC 2019
- [Caurant18] G. Caurant, The lighting technology of Detroit: Become Human, GDC 2018
- [Tatarchuk05] N. Tatarchuk, Irradiance Volumes for Games, GDC 2005
- [Lazarov13] D. Lazarov, Getting More Physical in Call of Duty: Black Ops II, SIGGRAPH 2013

References: Hair System

- [Bender et al. 2015] Position-Based Simulation Methods in Computer Graphics
- [Bridson and Müller-Fischer 2007] Fluid Simulation SIGGRAPH 2007 Course Notes
- [Gibou et al. 2002] A Second Order Accurate Symmetric Discretization of the Poisson Equation on Irregular Domains
- [Harris 2004] Fast Fluid Dynamics Simulation on the GPU
- [Kelager et al. 2010] A Triangle Bending Constraint Model for Position-Based Dynamics
- [Kim et al. 2012] Long Range Attachments - A Method to Simulate Inextensible Clothing in Computer Games
- [Kugelstadt and Schömer 2016] Position and Orientation Based Cosserat Rods
- [Losasso et al. 2008] Two-Way Coupled SPH and Particle Level Set Fluid Simulation
- [Macklin et al. 2014] Unified particle physics for real-time application
- [Macklin et al. 2019] Small Steps in Physics Simulation
- [McAdams et al. 2009] Detail Preserving Continuum Simulation of Straight Hair
- [Müller et al. 2006] Position Based Dynamics
- [Müller et al. 2012] Fast Simulation of Inextensible Hair and Fur
- [Petrovic et al. 2005] Volumetric Methods for Simulation and Rendering of Hair
- [Zhu and Bridson 2005] Animating Sand as a Fluid

References: Hair Shading & Rendering

- [Marschner et. al. 2003] Light Scattering from Human Hair Fibers
- [Hery et. al. 2007] Importance Sampling of Reflections from Hair Fibers
- [Zinke et. al 2008] Dual Scattering Approximation for Fast Multiple Scattering in Hair
- [Sadeghi et. al 2010] Efficient Implementation of the Dual Scattering Model in RenderMan
- [Sadeghi et. al. 2010] An Artist Friendly Hair Shading System
- [d'Eon et. al. 2011] An Energy-Conserving Hair Reflectance Model
- [Laine et. al. 2011] High-Performance Software Rasterization on GPUs
- [d'Eon et. al 2013] Importance Sampling for Physically-Based Hair Fiber Models
- [Karis 2013] Real Shading in Unreal Engine 4
- [Drobot 2015] Physically Based Area Lights
- [Yan et. al. 2015] Physically-Accurate Fur Reflectance: Modeling, Measurement and Rendering
- [Lagarde et. al. 2015] Moving Frostbite to Physically Based Rendering
- [Chiang et. al. 2016] A Practical and Controllable Hair and Fur Model for Production Path Tracing
- [Pharr 2016] The Implementation of a Hair Scattering Model
- [Karis 2016] Physically Based Hair Shading in Unreal
- [Yan et. al. 2017] A BSSRDF Model for Efficient Rendering of Fur with Global Illumination
- [Tafari 2019] Strand-based Hair Rendering in Frostbite
- [Taillandier, Valdes 2020] Every Strand Counts: Physics and Rendering Behind Frostbite's Hair

Special Thanks

- Torbjörn Laedre
- Julien Ignace
- Adrien de Tocqueville
- Antoine Lelievre
- Nicholas Brancaccio
- Rasmus Roenn Nielsen
- Tobias Franke
- Pierre Donzallaz
- Sebastien Lagarde
- Natalya Tatarchuk
- Kleber Garcia
- Timothy Lottes
- Unity's Tech Art team
- Unity's Lighting team (past and current members)